;login:
SYSTEMS ADMINISTRATION CONFERENCE (LISA VI) PROCEEDINGS
AUTUMN
1992

USENIX

CONFERENCE PROCEEDINGS

Systems Administration (LISA VI)

October 19-October 23, 1992
Long Beach, California

For additional copies of these proceedings contact

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Telephone: 510-528-8649

The price is $23 for members and $30 for nonmembers.

Outside the U.S.A. and Canada, please add
$12 per copy for postage (via air printed matter).

Past USENIX Large Installation Systems Administration Workshop and Conference Proceedings (price: member/nonmember)

Large Installation Systems Admin. I Workshop	1987	Phildelphia, PA	$4/$4
Large Installation Systems Admin. II Workshop	1988	Monterey, CA	$8/$8
Large Installation Systems Admin. III Workshop	1989	Austin, TX	$13/$13
Large Installation Systems Admin. IV Conference	1990	Colorado Spgs, CO	$15/$18
Large Installation Systems Admin. V Conference	1991	San Diego, CA	$20/$23

Outside the U.S.A. and Canada, please add $8
per copy for postage (via air printed matter).

ISBN 1-880446-47-2

Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.

USENIX Association

Proceedings of the Sixth Systems Administration Conference (LISA VI)

October 19-23, 1992
Long Beach, CA, USA

TABLE OF CONTENTS

System Administrator Training and Customer Satisfaction

Wednesday (4:30-5:00) **Chair: Jeff Polk**

Mass Host Configuration and Duplication

Thursday (9:30-10:30) **Chair: Trent Hein**

Mass Host Configuration and Duplication, II

Thursday (11:00-12:00) **Chair: Rob Kolstad**

System Administration Potpourri, I

Thursday (1:30-3:30) **Chair: Herb Morreale**

Distributed Software Management, I

Thursday (4:00-5:30) **Chair: Jeff Polk**

Distributed Configuration Management

Friday (9:30-11:00) **Chair: Pat Parseghian**

Monitoring System and User Problems

Friday (11:30-12:30) **Chair: Rik Farrow**

System Administration Potpourri, II

Friday (2:00-3:30) **Chair: Trent Hein**

Distributed Software Management, II

Friday (4:00-5:00) **Chair: Herb Morreale**

ACKNOWLEDGMENTS

PROGRAM CHAIR
Trent Hein, *XOR Network Engineering, Inc.*

PROGRAM COMMITTEE
Rik Farrow, *UNIX World*
Jeff Forys, *University of Utah*
John Hardt, *Martin Marietta Astronautics*
Rob Kolstad, *Berkeley Software Design, Inc.*
Herb Morreale, *XOR Network Engineering, Inc.*
Pat Parseghian, *AT&T Bell Laboratories*
Jeff Polk, *Berkeley Software Design, Inc.*

PROCEEDINGS PRODUCTION
Rob Kolstad, *Berkeley Software Design, Inc.*
Malloy Lithographing, Inc.
Carolyn Carr, *USENIX Association*

ALTERNATE TRACK COORDINATOR
Steve Simmons, *Industrial Technology Institute*

BOF COORDINATOR
Arch Mott, *Protocol Engines, Inc.*

TUTORIAL COORDINATOR
Daniel V. Klein, *USENIX Association*

TERMINAL ROOM COORDINATOR
Barb Dyker, *University of Colorado, Boulder*

VENDOR DISPLAY COORDINATOR
John Donnelly, *Seminars, Meetings, and Exhibits*

USENIX MEETING PLANNER
Judith F. Desharnais, *USENIX Association*

PREFACE

I have found LISA to always be the most amazing USENIX gathering. As a breed, system administrators seem to have a number of rather special traits. Among them, strong personalities and fastidiousness seem to always float to the top of the list. These are what have made LISA a raging success over the years, but are also what have forced LISA to be a carefully balanced technical and social gathering.

For the first time this year, LISA is a full-tracked 5 day conference, having grown from a small "workshop" in just 6 years. These proceedings are just a few pages longer than the most recent "general" USENIX conference (San Antonio). After looking at the quality of papers we have this year, I think I can safely say that it would be unfair for anyone to call LISA a "minor-league" conference from here on out. As system administrators, we can be all be proud of LISA, and I want to thank all of you reading this for making that happen.

Also, for the first time, the LISA (Large Installation System Administration) name is being kept for historical purposes only, and we are now welcoming system administrators from sites of all sizes, from one host to ten thousand. I feel that our outstanding technical program this year will address interests of this wide attendee base. Whether you're an old or new LISA attendee, please make the effort to share your views and make your interests known in the on-the-fly discussions that so often pop up in the lobbies or bars of a USENIX conference. The more our community interconnects, the stronger we'll be as a profession.

Organizing a conference like LISA requires an incredible amount of behind-the-scenes time and effort from an army of dedicated volunteers. Special thanks goes to all the members of my program committee, who have all been fantastic to work with. I would also like to thank our Rob Kolstad for acting as Board Liaison and Proceedings Typesetter, Arch Mott for volunteering as BOF Coordinator, Barb Dyker for organizing the terminal room, John Donnelly for planning the vendor displays, and Steve Simmons for organizing the alternate track. Last but most certainly not least, I would like to extend special thanks to Judy DesHarnais, who as our meeting planner has been infinitely patient and incredibly helpful to me as this year's program chair.

Trent R. Hein
Program Chair

AUTHOR INDEX

Effective Use of Local Workstation Disks in an NFS Network

Paul Anderson – University of Edinburgh

ABSTRACT

This paper presents an analysis of the NFS traffic generated by a typical group of Unix workstations. The relative advantages and disadvantages of holding different categories of data on local workstation disks are then discussed, in the light of these results. Finally, a program is described that automatically reconfigures a local disk, at regular intervals, to transparently hold copies of the most frequently used programs and data. Trace-driven simulations of this program, using the NFS trace data, and experience of the system in practice, show that considerable savings can be made in server and network traffic using comparatively small amounts of local disk space.

Introduction

A typical Unix network includes one or more fileservers that provide the bulk of the filestore for the workstations on the network. The use of central servers is usually more cost-effective in terms of storage and easier to administer than a highly distributed filesystem. However, many individual workstations also incorporate their own, much smaller disks, that are generally used to hold frequently accessed data, improving the workstation performance and reducing the load on the servers and the network.

Special filesystems, such as the Andrew File System [1] (AFS), are capable of utilising local disk space as a filesystem cache that automatically attempts to minimize remote file accesses. However, the most common remote filesystem for Unix workstations is probably Sun's Network Filesystem [2] (NFS) which provides no such inbuilt facilities. In a "traditional" NFS network, a small local disk would typically be configured to hold swap space and some, or all, of the vendor's Unix implementation. This scheme is easy to administer, because the local disk should only require updating for changes to the system configuration or new releases of the operating system. However, much of the traffic on a typical network involves access to home directories and locally-maintained software. These data change more frequently and are much larger and hence more difficult to manage when they are distributed across local disks. Many installations attempt to store some local software and user files on workstation disks, but the choice of which data to hold is often arbitrary and measurement of the actual effectiveness is difficult.

By monitoring the NFS traffic from a group of workstations, it is possible to categorize the filesystems being accessed and predict the traffic savings that could be obtained by storing different filesystems on a local disk. This allows the effectiveness of simple disk allocation schemes to be compared. For certain types of data (read-only program and data files), a more detailed analysis, at the individual file level, shows which files are being accessed most frequently and the savings that could be obtained by holding specific subsets of these files locally. By examining the access times of such files, it is possible for a program to regularly re-configure a local filesystem so that it always holds an optimal subset.

NFS Traffic Analysis

The Department of Computer Science network at the University of Edinburgh includes several hundred heterogeneous workstations spanning several subnets. Each subnet is self-contained in terms of workstation services, such as booting, base operating system and local software, but the automounter AMD [3] provides a global namespace that presents a uniform filesystem across all the workstations, using NFS as the remote access protocol[4, 5].

A typical selection of workstations on a single subnet was chosen as a representative sample for the purposes of traffic analysis. These consisted of 26 assorted Sun workstations running SunOS 4.1, classified as follows[1]:

personal 23 personal workstations, 4 with their own swap disks.

public 1 public multi-user machine with a local disk holding the base operating system and local swap space.

compute 2 compute servers with their own disks, used for swapping only.

These workstations serve a computer science research laboratory, running applications such as text processing, mail, and program development. The

[1] The fileservers on the subnet are not included, since the traffic to the servers will be exactly that traffic generated by the workstations.

environment is similar to many academic and research installations, and, although there may be significant differences in usage patterns for teaching or commercial networks, many of the following results should still be appropriate.

The programs *rpcspy* and *nfstrace* [6] were used to collect data on all NFS transactions from the above workstations over a period of one month. These data were analyzed by several *perl* [7] scripts that divided the traffic into the following categories by examining the NFS filehandles present in the transactions:

home Access to user's home directories.

local Access to locally maintained software.

root Access to the workstation's *root* partition.

usr Access to the workstation's *usr* partition.

swap Access to the swap file.

misc Miscellaneous access to remote filesystems. This includes print spool directories, network-wide temporary directories and remote traffic to other domains.

As would be expected, the amount and distribution of traffic between the categories varies considerably between individual workstations, and from day to day. This is owing to the changing nature of the individual's work; a particular user might even be absent for a week, leaving a workstation idle. However, it is the average figures that reflect the total impact of all the workstations on the network and the servers.

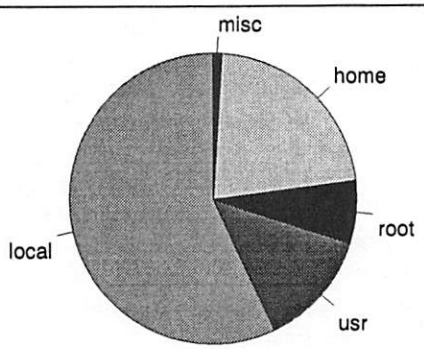


Figure 1

On the diskless workstations, swapping traffic accounted for an average 30% of the total NFS traffic during the month. For individual workstations, this varied between 10% and 40%, obviously depending on the type of work being performed by the user, and on the real memory available[2]. It was noticeable that users with heavy compute requirements are not necessarily those that generate most swap traffic, since they often tend to use remote compute servers for large jobs.

The public machine and the personal workstations showed a roughly similar distribution of the remaining traffic (Figure 1), although the balance between local software and home directories varied considerably; in the extreme case of the two compute servers, these two values were actually reversed. This appears to be due to compute-intensive programs accessing large data files from user's home directories, and paging off users own program binaries.

Standard Configurations for a Local Disk

A conventional allocation of space on a local workstation disk would probably include *swap*, *root* and *usr* filesystems.

In general, local swap space is almost always effective, although the actual amount of disk space required, and the amount of traffic saved on any individual workstation, will vary greatly. The above situation is probably typical, representing an average traffic saving of around 30%, with a swap space of 20-30Mb.

Local storage of the *root* filesystem is also efficient, occupying about 5Mb of disk and saving 8% of the traffic, but the total saving is small and the use of local root partitions is more difficult to manage.

In those situations, where users rely largely on the software supplied with the vendor's operating system, local storage of the entire *usr* filesystem could produce a substantial saving in network traffic; this may also provide some degree of self-sufficiency for the workstation in the event of a server failure. In many large installations, however, users require access to a much wider range of software; in the above example, most users rely on local software even for their shells and window systems, so a local copy of the *usr* would only generate a saving of about 10% for 130Mb of disk space, and would not provide any useful degree of resilience.

Public Files and Software

Network wide public programs and read-only data files (such as fonts) account for a large proportion of the network traffic. For the above sample, most of this traffic is concentrated on the separate filesystem holding locally maintained software (*local*), although for many installations, a larger proportion might be generated from the standard *usr* filesystem. However, both these filesystems have exactly the same characteristics and it would seem reasonable to hold local copies of frequently used public programs and data files, whatever their origin. Such files are not normally written to by the workstation and it is usually sufficient to update the local copies from some master copy at relatively

[2]Mostly 16Mb in this sample.

infrequent intervals - for example nightly, or simply on demand. This seems to be common practice and several schemes have been documented, using widely available software[8, 9, 10, 11]. In general, however, there will be far too much software available on the network for copies of everything to be held locally (our network has over 700Mb of *local* software and 130Mb in *usr*), so some subset of the available software needs to be selected. A number of schemes have been proposed[10, 12, 13, 14] for organising master copies of the software so that sensible subsets (usually corresponding to individual applications) can be extracted and selectively copied onto a local disk. Some of these, such as *Depot*[12] and *lfu*[10] include software which can replace certain subsets of files with symbolic links. This would allow a local filesystem to store actual replicas of some files, and links to remote copies of other files, thus permitting files to be transparently migrated on and off the local disk by changing links into file copies and visa-versa.

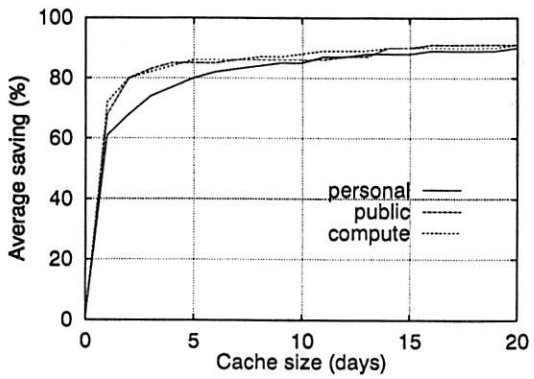


Figure 2

This type of scheme has the potential to be very effective; for our network sample, Figure 2 shows the percentage of *local* traffic[3] which could be saved by holding copies of just those files accessed in the last N days. These results are surprisingly consistent across the different types of workstation and show that nearly 90% of network traffic from the *local* software occurs to files that have been accessed in the last two weeks; it seems that only a small fraction of the total available software is actually in use at any one time.

Figure 3 shows the local disk space required to hold all files accessed in the last N days, implying that 90% of the *local* traffic on a personal workstation can be saved for only 40Mb of disk space, *providing* that the appropriate files can be chosen. Unfortunately, the optimal fileset for local storage can be very different on different workstations and obviously varies with time; this is reflected in the larger quantities of disk space required by the public machine and the compute servers, which are used by many different people. Making a reasonable choice manually is very difficult; for example, out of the hundreds of fonts for the window system, each user will have a personal preference and will use only a small, but different, fraction of those available.

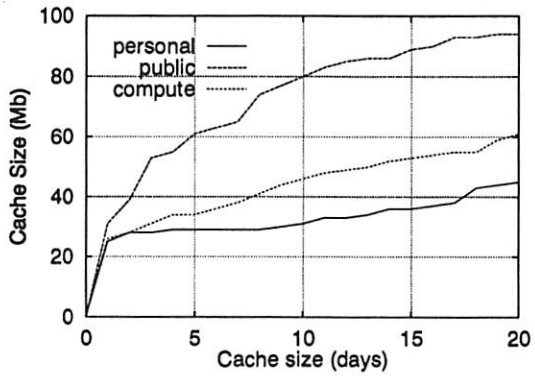


Figure 3

File Caching

The above results suggest that a file caching scheme which stored only the recently used files on a local "cache" disk, could be very effective. Other work on general-purpose file caching schemes[15] has confirmed the effectiveness of this technique, but it is unlikely that any general-purpose cache could be effectively implemented without extensive modifications to the kernel filesystem. However, this particular category of files has a number of properties which simplify the problem considerably, and many of the necessary mechanisms are already in place:

- Immediate reflection of master changes into the cache is not normally critical.
- The local copies are normally *read-only* as far as the workstation is concerned.
- A file can be moved out of the local cache, simply by replacing it with a symbolic link to a network-mounted copy of the file. Conversely, the a file can be encached by replacing the link with an actual copy of the file.
- If one of the available distribution schemes is already being used, then cache re-arrangement can conveniently occur at the same time as the local files are updated from the master.

The only remaining problem is to select the optimal set of files to be held in the cache, and for the software distribution program to re-arrange the local disk so that these files are held locally, and other files are replaced with links back to the network copies.

[3]Local software has been emphasised in this discussion because it forms a much larger proportion of the traffic in our network sample, but there is no reason to expect that the results from the *usr* filesystem would be significantly different.

It should be quite possible for the selection of the optimal fileset to be performed independently of the cache update, and this is probably the cleanest solution. However, if the update program is already traversing the filesystem to identify out-of-date files, it is very easy for the same program to collect information about the previous usage of these files, and this is the approach adopted below.

A Caching Version of the *lfu* Program

The Computer Science department network uses nightly runs of the *lfu* [10] program to update copies of the *local* software onto multiple servers and onto the larger workstations. This program simply performs a selective copy of files from the master directories (mounted over NFS) onto the local disk. Those files which have been updated on the master are re-copied, and manually selected subsets of files can be blocked, or replaced with symbolic links to copies elsewhere on the network.

To investigate the possibilities of an automatic caching scheme, *lfu* was modified to:

- Collect information on the previous usage of the files and links on the local disk.
- Apply an algorithm to each file, computing a *cache score* intended to reflect the desirability of holding a local copy of the file.
- Re-arrange the local disk to hold as many files as possible with the highest cache score (replacing other files with symbolic links).

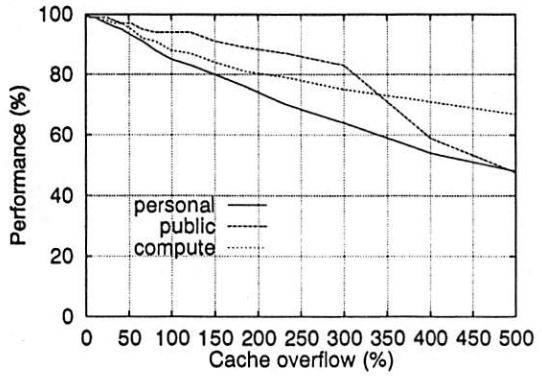


Figure 4

As each file is examined, the "last access time" (*atime*) of the file is used to record whether or not the file has been used since the last update. This allows a running history of the file usage over the last two weeks to be constructed and maintained on the disk. For an infinite-sized cache, this would be sufficient and the 90% saving indicated by the simulations could be expected simply by holding local copies of all files used in the last two weeks. In practice, the cache disk has a fixed size, and there needs to be some way of deciding which files to remove from the cache if it becomes full. Two *perl* scripts were used to perform full simulations of the cache behaviour, on the NFS trace data, to investigate the performance of different fixed cache sizes, with different algorithms; *least-frequently-used* and *least-recently-used*. The daily inspection of the file *atime* provides only a very coarse measure of the file usage, but the following results show that this is more than adequate in practice; Figure 4 gives the simulated performance degradation, for different levels of cache overflow, using the *least-frequently-used* algorithm (100% represents the performance of an infinite cache).

The trace-driven simulations show a very similar performance for the two algorithms, indicating that the most *frequently* used files are also the most *recently* used files, and that the choice of algorithm is not critical.

The *least-frequently-used* algorithm was implemented in *lfu*, with the additional option of locking certain files into the cache, or varying the priority applied to the cache score. A number of personal workstations and a Sun 690/MP compute/multi-user machine were configured to run this caching scheme. The workstations were allocated 70Mb and the 690/MP, 120Mb. This includes the 40Mb (90Mb) indicated by Figure 3, and 30Mb of overhead for the directories and symbolic links needed to reference the 43,000 files on the master. Figure 5 shows the real performance of the cache on the 690/MP.

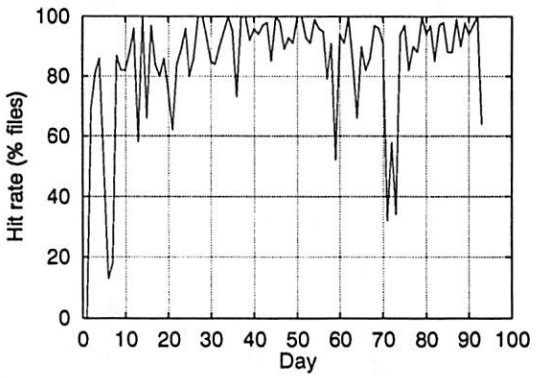


Figure 5

In this case, the data were gathered from the live *lfu* program and shows the percentage of different files, on each day, which were accessed from the cache. Although this is not directly comparable with the simulation data (which shows the percentage of blocks transferred) it provides an indication that the cache is performing as expected. Examination of the actual files being decached, shows that the sizing of the cache is also approximately correct; most files being removed from the cache having been used on no more than one day in the last two weeks.

Some Practical Issues

A nightly update of the cache generates some network traffic that is not included in the above

figures; this could be a problem for sites running jobs that require significant amounts of network bandwidth during the night. However, the bulk of this traffic would already be generated by any conventional distribution scheme and is concerned with identifying files that need to be updated. Examination of typical *lfu* updates (which are not particularly efficient), using *etherfind*, [16] shows that about 1KB of network traffic is generated for each file scanned, yielding a total of about 40Mb for the 40,000 files on the sample system[4] and the extra traffic necessary to adjust the cache, amounts to only an additional 5-10%. Careful scheduling of the nightly updates (which normally take about half an hour each) can usually avoid network congestion and interference with other background jobs, such as backup.

Holding only the most recently used files, means that the local disk provides no real resilience against server failure; certain very important files (such as files required only at boot time) may not be used regularly enough to appear in the cache. The graph in Figure 2 does indicate that a reasonable degree of resilience may not be achievable without holding copies of all possible files, since there appears to be a constant 10% of "new" files used, on average, each day. A more successful solution to this problem is to provide several alternative servers carrying complete sets of all the files (This redundancy is comparatively easy to arrange, using AMD). It is possible however, to "wire down" the contents of the cache at any point (for example, after a reboot), or to manually specify files which should be locked in the cache for resilience purposes.

The coarse-grain measure of file usage can sometimes cause the cache to be "flooded" by a single unexpected access to a large number of infrequently used files. For example, a user running the command "`grep FOO *`" in `/usr/include` will cause the atime of all files in the directory to be updated, possibly raising their cache score above more regularly used files[5]. The use of a *least-frequently-used* algorithm (rather than *least-recently-used*) and a threshold of two days usage for initial entry into the cache, prevent nearly all of these problems in practice.

Several technical problems prevent the *atime* of files from always giving a reliable indication of the file usage. In particular, the *atime* of continuously running programs is not continuously updated, and the *atime* of files which are exported over NFS is not always correct. Symbolic link *atimes* are also updated rather too easily; for example, by the command "`ls -l`". *lfu* uses a number of heuristics to deal with the worst of these cases and they do not appear to be a problem in practice.

Home Directories

Home directory access is responsible for a large percentage of the traffic from most workstations. If these directories are held on a central fileserver, considerable savings could be expected by moving them onto a local disk. For the above sample, Figure 6 shows the percentage of this traffic that could be saved by transferring user's home directories onto their own workstations, and the amount of disk space that would be required in each case[6].

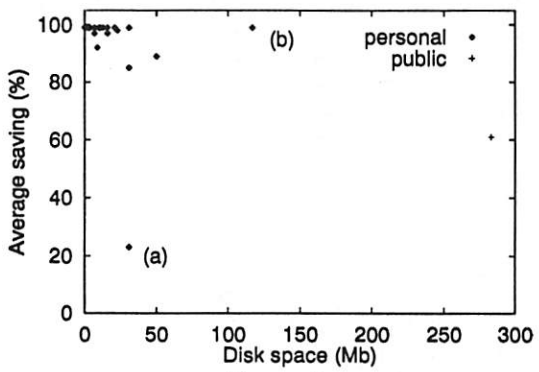


Figure 6

In most cases, virtually all of the home directory traffic can be eliminated by allocating about 50Mb of disk space to the user's home directory[7]. However, this has a number of drawbacks:

- The amount of space required is difficult to predict and may vary greatly between users. In the above example, one user (b) requires much more than 50Mb and many users require much less. Reallocation and efficient use of this space is difficult.
- The directory must still be available from elsewhere on the network which means that the workstation must export the filesystem over NFS. Integrating filesystems from large numbers of different workstations and arranging backups is more difficult to manage.
- The savings are greatly reduced in any situation where a user regularly works on more than one machine; for example, in a pool of ten compute servers, or public machines, which are used at random, the saving will be less than 10%.

[4]Clearly, if a lot of files require updating because the masters have changed, then this figure will increase.

[5]This can be observed at several points in Figure 5, which displays the cache hit rate as the percentage of different files, rather than network traffic. The low hit-rate around day 71 was generated by a user browsing a large number of rarely used fonts.

[6]Users without their own workstation are transferred onto the public machine.

[7]The one workstation showing a particularly low saving in the above example (a) represents a machine that was being used by several different users for a particular package with node-locked licensing.

When a caching scheme is being used for public software, then it may be beneficial to encourage users to export copies of their own software along with the other public software. This is especially useful if the software is heavily used and/or frequently used by other people, when it provides the performance benefits of caching, together with resilience for other users of the program.

Some Possible Developments

Most existing distribution schemes work well when they are updating small numbers of files on a large number of machines, or a large number of files on a small number of machines. The above caching scheme is appropriate for most workstations with their own disk, however small the disk, so it makes unusual demands on the distribution system by requiring very large numbers of master files to be scanned by every workstation, even though the number of files actually transferred is usually very small. This scanning process is by far the most costly factor in the nightly update operation and some type of update program which could simultaneously broadcast the information to several clients would considerably reduce server and network load.

The current implementation links only files, and always copies directories. This means that there may be very large directory hierarchies, that are never accessed by a particular machine, but still occupy significant amounts of space on the local disk for the directories themselves and the symbolic links that they contain. In theory, such a hierarchy could be replaced by a single symbolic link to the remote root of the hierarchy, but in practice, it is difficult (perhaps impossible) to identify individual workstation access to files in such a hierarchy and to arrange for its expansion when a decision is made to encache one of the files that it contains.

Conclusions

Clearly there will be some variation in usage patterns between sites, that will affect the optimal configuration for local workstation disks. There are also a number of trade-offs to be made; for example, the trade-off between efficiency and ease of management involved in the placement of home directories. The degree to which traffic can be saved[8] by caching a small number of well-chosen files on a personal workstation is, however, surprising and a number of conclusions would seem to be generally applicable:

- Holding *swap* and *root* partitions on a local disk is generally effective.
- Holding full copies of the *usr* filesystem may not be worthwhile in many cases.
- Caching of public files is very efficient in terms of reducing the network and server load. This does involve additional administration and nightly network bandwidth, but not much more than existing schemes which update fixed sets of files onto a local disk.
- Holding local copies of home directories is normally effective only for personal workstations, or machines that are used significantly more often by an individual user. It is likely to be difficult to match space requirements to available disk space though, and some compromise such as providing the user with both a remote, and a local directory may be necessary.

For a typical personal workstations from the survey, the following configuration would be an effective way of organising a local 150Mb disk:

- 25Mb *swap*, saving 30%
- 5Mb *root*, saving 5%
- 70Mb *local* cache, saving 90% of 41% = 36%
- 50Mb *home*, saving 15%

Thus providing a total saving of over 80% of the network and server load generated by the workstation.

Optimization of local disks on public machines and compute servers is more difficult; caching of public files is still effective, but the home directory traffic becomes a more significant proportion of the total traffic and this is more difficult to place and less effective than on a personal machine.

Acknowledgments

Thanks are due to Matt Blaze for the use of the *rpcspy* and *nfstrace* programs, and to the system staff in the Computer Science Department for their suggestions and assistance during development and testing of the *lfu* program.

Availability

The caching version of *lfu* is available for anonymous ftp from `ftp.dcs.ed.ac.uk` in the directory `pub/lfu`. The documentation file in this directory also includes the references[10, 4, 5].

References

1. Edward R Zayas, "AFS-3 Architectural Overview," in *AFS-3 Programmer's Reference Manual*, Transarc Corporation, September 1991.
2. R Sandberg, D Goldberg, S Kleiman, D Walsh, and B Lyon, "Design and implementation of the Sun Network File System," *Proceedings of Usenix Conference*, Summer 1985.
3. Jan-Simon Pendry, *AMD – An automounter*, Department of Computing, Imperial College, London, May 1990.
4. Paul Anderson and Alastair Scobie, "The Local UNIX directory hierarchy," CS-TN-21,

[8] Note that this does not necessarily imply an improved performance for every workstation; network access to fast disks on a lightly loaded server can be faster than slow local disks.

Department of Computer Science, University of Edinburgh, Edinburgh, August 1991.
5. Paul Anderson, "Installing software on the Computer Science Department network," CS-TN-24, Department of Computer Science, University of Edinburgh, Edinburgh, August 1991.
6. Matt Blaze, *NFS Tracing by passive network monitoring,* Department of Computer Science, Princeton University.
7. Larry Wall, *Perl – Practical Extraction and Report Language.*
8. Sun Microsystems, "rdist (1)," in *SunOS reference manual*, Sun Microsystems, 1990.
9. Steven Shafter and Mary Thompson, *The SUP software upgrade protocol,* Carnegie Mellon University, September 1989.
10. Paul Anderson, "Managing program binaries in a heterogeneous UNIX network," *Proceedings of LISA V Conference*, pp. 1-9, Usenix, 1991.
11. Bjorn Satdeva and Paul M Moriarty, "Fdist: A domain based file distribution system for a heterogeneous environment," *Proceedings of LISA V Conference*, pp. 109-126, Usenix, 1991.
12. Wallace Colyer and Walter Young, *Depot: A tool for managing software environments,* Carnegie Mellon University, April 1992.
13. John Sellens, "Software maintenance in a campus environment: the Xhier approach," *Proceedings of LISA V Conference*, pp. 21-28, Usenix, 1991.
14. Kenneth Mannheimer, Barry A Warsaw, Stephen N Clark, and Walter Rowe, "The Depot: A framework for sharing software installation across organizational and UNIX platform boundaries," *Proceedings of LISA IV Conference*, pp. 37-46, Usenix, 1990.
15. Matt Blaze and Raphael Alonso, *Long-term caching strategies for very large distributed file systems,* Princeton University.
16. Sun Microsystems, "etherfind (8)," in *SunOS reference manual*, Sun Microsystems, 1990.

Author Information

Paul Anderson graduated in Pure Mathematics from the University of Wales in 1977. He taught Mathematics and Computer Science at the North East Wales Institute of Higher Education until 1984 when he became system manager for the Institute, establishing a new computer centre and software development team.

In 1988 he moved to the University of Edinburgh as Systems Development Manager with the Laboratory for the Foundations of Computer Science, where he is currently managing the laboratory network and working with other system managers to develop the computing facilities. Paul can be reached by mail at the Laboratory for the Foundations of Computer Science; Department of Computer Science; University of Edinburgh; King's Buildings; Edinburgh; EH8 3JZ; U.K. Reach him electronically at paul@dcs.ed.ac.uk.

Optimal Routing of IP Packets to Multi-Homed Servers†

Karl L. Swartz – Stanford Linear Accelerator Center

ABSTRACT

Multi-homing, or direct attachment to multiple networks, offers both performance and availability benefits for important servers on busy networks. Exploiting these benefits to their fullest requires a modicum of routing knowledge in the clients. Careful policy control must also be reflected in the routing used within the network to make best use of specialized and often scarce resources. While relatively straightforward in theory, this problem becomes much more difficult to solve in a real network containing often intractable implementations from a variety of vendors.

This paper presents an analysis of the problem and proposes a useful solution for a typical campus network. Application of this solution at the Stanford Linear Accelerator Center is studied and the problems and pitfalls encountered are discussed, as are the workarounds used to make the system work in the real world.

Introduction

In a simple network composed of a collection of machines connected to a single multi-access network, such as an Ethernet, routing of IP packets is simple since each host can talk to every other host on the network. Introduction of a second network, connected via a router, adds only slightly to complexity – any non-local hosts must be reached via the router. This can be trivially implemented by adding a *default* route on each host pointing to the router's adjacent port.

Introduction of the second network and the router does serve to segregate traffic and thus reduce network loads, but it also creates some new problems if hosts on both networks share one or more large servers. If the router goes down, one network loses access to the server. If the network to which the server is attached goes down, perhaps for maintenance, both networks lose access to the server. If the two networks serve two different user communities with very different work habits, this may make it difficult to find a time when maintenance can be done on the critical network, and perhaps costly as well if overtime is required.

Problems with NFS and routers

If the server is an NFS fileserver, another more subtle problem may appear. NFS uses 8 kilobytes as its default packet size, a value which parallels the largest possible block size in a BSD filesystem. Ethernet, on the other hand, has a Maximum Transfer Unit (MTU) of 1500 bytes. In order to pass an 8 KB packet over an Ethernet, IP must fragment this large packet into a number of smaller packets, which are re-assembled when they reach the destination. If the network is congested the sender may delay between each packet. Ideally the destination is eagerly awaiting the packets and will have no trouble accepting all of the fragments. An intermediate device such as the router, however, may not be able to accommodate all of the incoming traffic and may simply drop some of the fragments. The destination will eventually decide that some fragments are never going to arrive and will purge the entire packet. The NFS client software, with no knowledge of this fragmentation, simply sees a slow or unresponsive server and will reissue the request, at some point also issuing the infamous `NFS server not responding` diagnostic.[1]

One can certainly reduce the NFS packet size by adjusting the **rsize** and **wsize** parameters of the mount, but this could have a substantial performance impact of its own, and requires knowledge of the network topology by each client to decide whether or not the smaller values should be used. Recent versions of AMD [2] address the latter issue through the **wire** selector, though the other problems still remain.

Multi-homed servers

Instead of trying to work around the problems introduced by addition of the router, a better approach is to avoid them wherever possible by connecting the critical server to both networks. One

†This work supported by the United States Department of Energy under contract number DE-AC03-76SF00515, and simultaneously published as SLAC PUB-5895.

[1] A discussion of this problem is contained in [1]. "Appendix C: NFS Problem Diagnosis" of that document details how to determine whether or not dropped packets are a problem for a particular system.

could eliminate the router altogether and use the server as a router, and some server vendors suggest this. This is probably going a bit too far unless there is little traffic between the two networks. Even relatively inexpensive routers are better at this job than most fileservers, and fileservers burdened with many incidental tasks tend to become poor fileservers. Using a router in parallel with a multi-homed server, as in Figure 1, provides the best service both for traffic between the networks and for access to the server.

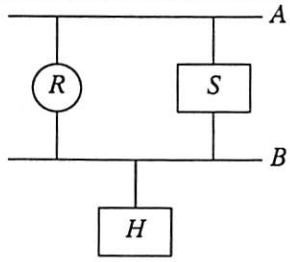


Figure 1: Simple network with multi-homed server

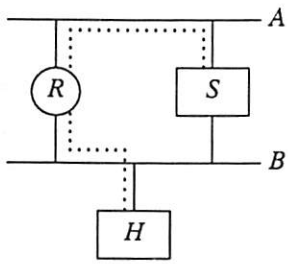


Figure 2: Path using default route to S_A

Now the challenge becomes one of getting the hosts to talk to the server directly rather than via the router. If the host in Figure 1 refers to the server using the IP address on network B (or for brevity of notation simply S_B), all is well. If, however, the host uses S_A, it will see that the address is not on its local network and will use the default route to choose the router as the gateway to the final destination, as illustrated in Figure 2. Besides placing unnecessary traffic on network A, this affects availability by introducing dependencies on both router R and network A. It also raises the specter of packet fragmentation problems.

One could simply use different names for S_A and S_B, and this is what some vendors recommend [3]. Users probably won't remember the difference, however, even if they understand the network topology sufficiently. NFS configuration also becomes more complicated, which may be a substantial consideration for a site with many machines and servers.

Another approach is to use the same name but to somehow resolve it to the closest address for each machine. This host-specific name resolution can easily be accomplished via `/etc/hosts`, though this doesn't scale well to a large network. Sun's Network Information Service (NIS, formerly known as Yellow Pages) [4] is commonly used, for better or worse, to assist with maintenance of the hosts database, but it provides for only one address per host name across the entire network. The Domain Name Service (DNS) [5] provides for multiple addresses but no preference for one over another, though some implementations do provide a facility for sorting multiple addresses [6].

Host routes

With resolution of a single server name to the best address being eliminated as a viable option in many cases, the focus turns to what can be done if a host chooses the "wrong" address for the server. A host's routing table can certainly be more sophisticated than a simple default route for all non-adjacent addresses. In particular, a *host route* may specify a different route to one particular destination host than would be used for other hosts on the same network. Figure 3 illustrates the host's view of this.

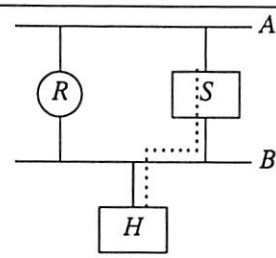


Figure 4: Path using host route to S_A

Here, S_B is the gateway to S_A, while R_B is used as the gateway to other hosts on network A. Thus

```
$ netstat -r
Routing tables
Destination  Gateway     Flags  Refcnt  Use   Interface
localhost    localhost   UH     2       38    lo0
default      ROUTER-B    UG     0       3     le0
A            ROUTER-B    UG     3       296   le0
SERVER-A     SERVER-B    UGH    8       758   le0
B            HOST        U      22      516   le0
```

Figure 3: Routing table with host route to S_A

packets sent by the host to the server's "wrong" address will still go directly to the server, as in Figure 4. Some additional processing is required on the server but for any modern server this should be insignificant.

Dynamic routing

For the network presented thus far, static routing is adequate (if a bit tedious) to implement everywhere. For a real network with many servers and perhaps multiple routers, maintenance of the routing tables on a multitude of hosts using static routes would be prohibitive. Fortunately, dynamic routing protocols exist which permit network devices to discover routes on their own and which enable them to adapt to changes in network topology, planned or otherwise.

Three different types of participants in the routing process have been encountered in this discussion, each with different needs and capabilities. Starting with the simplest, they are

- A workstation or other host *H* with only a single interface. It silently listens to routing information on the network and updates its own routing tables as necessary. Having only one interface, it can perform no useful gateWaying and thus has no need to advertise any routing information to others.
- A server *S* which has multiple interfaces, i.e., it is multi-homed. Like a workstation, it listens to routing information on the network and updates its own routing tables. (It should also be able to choose the best interface for outgoing traffic when confronted with a choice.) In addition, it acts as a gateway for packets addressed to one of its interfaces that are received via a different interface, and it must advertise routing information (host routes) so other hosts know of this service. As a matter of policy it does not forward packets between two networks and thus should not advertise any network routes.
- A router *R* which gateways packets between networks on behalf of other hosts. Logically, it is nothing more than a host with two or more interfaces, but without the policy restriction against passing packets between networks. In practice it will probably be a specialized device and may thus be less flexible in how policies may be implemented. Since it is willing to pass packets beyond itself it must also advertise network routes, including ones learned from other gateways and not just the host routes which a server advertises.

Choosing a routing protocol

As with editors, shells, and mailers on UNIX, an abundance of routing protocols exist for IP. Fortunately, most can be eliminated as being irrelevant to the task of *interior routing* – routing within a single network. When selecting from the available interior routing protocols, the requirement that hosts be able to listen to the protocol is the most stringent. Nearly every TCP/IP package includes support for RIP, the Routing Information Protocol [7], often via the BSD **routed** utility. Relatively few support any other protocol.

Newer and perhaps better protocols do exist, and portable software is available to implement many of them. Cornell University's **gated** [8] is a notable example of such software. A better choice of protocol might be cisco's Gateway Discovery Protocol (GDP) [9] or the ICMP Router Discovery Protocol [10], for which several UNIX implementations are freely available in source form. At best, this approach suggests a lot of porting effort for a typical installation with a multi-vendor environment. More likely, there are at least some devices which are "black boxes" and thus not amenable to this solution. FastPath gateways for AppleTalk, which support only RIP [11] and which cannot be enhanced with new protocols by the user, are a good example.

For better or worse, RIP appears to be the only reasonable choice available at this time if one wishes to dynamically communicate routing information to as wide a variety of hosts as possible. (Even this doesn't cover some hosts, which only support static routes to a default gateway, such as DOS machines using FTP Software's PC/TCP [12].) The question, then, is whether or not RIP is adequate for the job.

Suitability of RIP

A RIP packet is broadcast on each attached network by a router or other device which has routes to advertise. For each route, the packet includes a destination, which may be a host, a network, or a special value for default routes, and a *metric*, which indicates the quality of the route. The metric is an integer, customarily representing the number of *hops* or intermediary nodes required to reach the destination. For the network in Figure 1, *R* broadcasts on network *B* a packet advertising a route to network *A* with a metric of 1. If a host sees a route with a lower metric than one it is already using, it switches to the new route. In addition, a host route to a destination on a given network will be replaced, or *subsumed*, by a network route to the destination's network which has an equal or lower metric.

Nothing in RIP requires that the metric be a hop count, however. The only restriction is that values greater than or equal to 16 are considered to be *unreachable* – a RIP metric of 16 is commonly discussed as being "infinite." This is useful, since a host route to S_A via S_B with a metric of 1 would be subsumed by a network route to *A* via R_B with a metric of 1. If the network route instead has a metric of 2, the subsumption rule does not come into

play and the host route is used, which is exactly what is desired.

It is evident that RIP is capable of conveying the necessary routing information. The remaining problem is whether or not the servers and routers can be made to generate the desired RIP advertisements. A server should advertise host routes to each of its interfaces with a metric of 1; it should not advertise any other routes since routers will handle traffic between networks. A router must simply *inflate* its metrics to 2 instead of the usual 1.

Policy control in RIP implementations

The BSD **routed** utility doesn't offer much in the way of control by external policies. In particular, one can keep it from generating any advertisements, but there is no way to have it generate host routes to its own interfaces while suppressing other advertisements, nor is there a way to inflate metrics.

Cornell's **gated** offers a great deal of tailorability in addition to its support for a number of routing protocols. A **gated** configuration which will implement the desired policy for servers is illustrated in Figure 5. The first section in this configuration enables RIP. The second declares host routes to each of the server's addresses, which are not installed in the kernel routing table but which provide the necessary host routes for the RIP advertisements. The third section restricts RIP from advertising routes learned from other servers and routers, and causes the static host routes to be advertised with a metric of 1.

Few vendors distribute **gated** but, fortunately, porting is not as big a concern as for most hosts, since servers typically come from few vendors, compared to the multitude of sources for other hosts on the network. The sources from Cornell readily build on a variety of common UNIX platforms, and porting to VAX/VMS is not an issue since TGV provides **gated** with their MultiNet TCP/IP software. Porting to other non-UNIX platforms would probably be more difficult.[2]

The necessary configuration for routers is simpler, yet much more difficult to generalize, since there is no common ground between vendors for configuration options. All of SLAC's routers are from cisco Systems. Because ciscos are quite popular and constitute the majority of the author's experience, only the cisco configuration will be considered.

[2]Some creativity might be required to implement the necessary policies on peculiar servers. For particularly intractable systems, a hacked version of **routed** might prove more fruitful than trying to port **gated**. The author has so far successfully avoided the chore of porting **gated** to VM/CMS, though this necessity looms ominously.

```
# Enable RIP

rip yes;

# Static routes to our own interfaces -- handles for advertisements

static {
    host S_A gateway 127.0.0.1 noinstall;
    host S_B gateway 127.0.0.1 noinstall;
};

# Advertise only host routes to our various addresses

export proto rip {
    proto rip restrict;
    proto static {
        all metric 1;
    };
};
```

Figure 5: **gated** configuration for server *S*

```
! metric inflation -- match all addresses
access-list 42 permit 0.0.0.0 255.255.255.255

! configure RIP routing process
router rip
offset-list 42 out 1
```

Figure 6: Configuration fragment for cisco router

Fortunately, the necessary metric inflation is easily accomplished using the configuration fragment in Figure 6. The key here is the **offset-list** command, which causes 1 to be added to the metric of advertised routes to destinations matched by access list 42, which simply matches all destinations.

One difficulty is that cisco routers do not currently understand host routes. They will be implemented in the upcoming 9.1 release of the cisco software. Until then, the impact is not too great since traffic that does not require a router will not be affected. Only packets which must already pass through one router run the risk of passing through more routers than necessary, and many of the benefits of a direct path disappear at the first router hop. Reliability should be handled for the most part by redundant router paths, which are needed to provide reliable service for other traffic anyway.

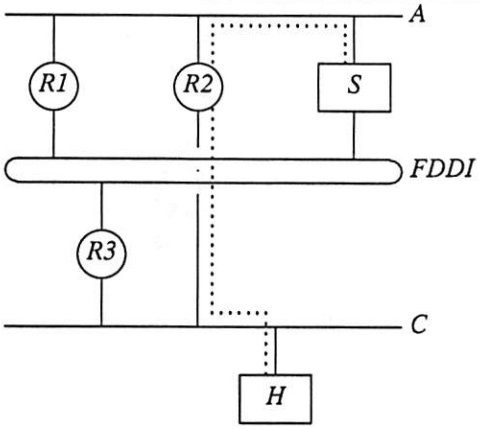


Figure 7a: Route with single Ethernet hop

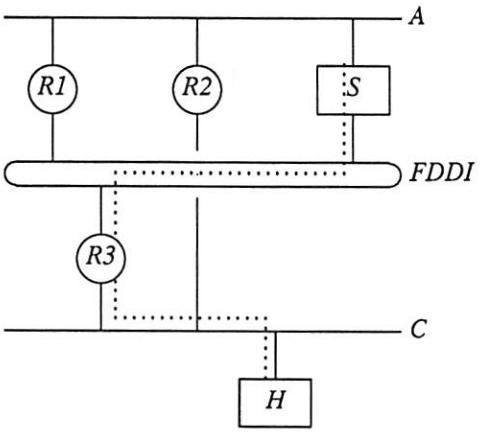


Figure 7b: FDDI plus host route (preferred)

Figure 7: Alternate routes from H to S_A

Scalability of RIP solution

With the addition of another network and another router, the metric for the route to the distant network increases to 4, with each of the routers making a contribution of 2. As the network grows larger, with an increasing number of router hops between the most distant portions of the network, the "infinite" metric of 16 will be approached. With a metric value of 2 for each router this implies a maximum path or *network diameter* of seven routers. This is probably beyond the reasonable limit for RIP due to inherent limitations in the protocol.

Introduction of another type of network adds a new level of complexity to the problem. So far all networks have been considered to be equal. If one of the networks is an FDDI ring, while the others are Ethernets, preference should almost surely be given to the faster and higher-bandwidth FDDI network, as depicted in Figure 7. RIP is still able to describe this topology, though the limits of scalability are much closer. The optimal route, in Figure 7b, has a metric of 3 – 1 for the host route from S_{FDDI} to S_A and another 2 added by the traversal of router $R3$. Therefore the Ethernet hop via $R2$ must have a metric of at least 4. This brings the limit on network diameter to only three routers, or four if one pair is connected via the FDDI network.

Routes between multi-homed servers that share two common networks are also a concern. In Figure 8a, either of the available routes is as good as the other, but if one of the networks is superior to the other, as in Figure 8b, it should be the preferred path between the servers.

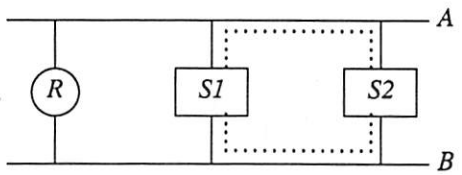


Figure 8a: Either route may be chosen

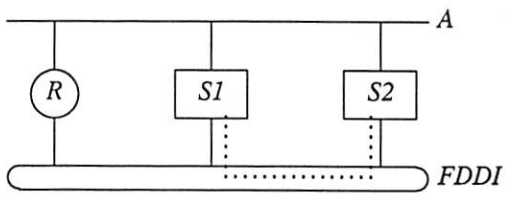


Figure 8b: Best route is via *FDDI*

Figure 8: Network preference amongst host routes

While the theory of using RIP to support both multi-homed hosts and a heterogeneous collection of networks may be interesting, it clearly shows signs of pushing the abilities of the protocol to its limits. Moreover, when one moves from theory to

implementation, the problem becomes intransigent. The most apparent obstacle is that cisco does not provide any means of inflating the metrics of some routes to 2 while others are inflated to 4. This alone eliminates the possibility of solving the problem with only RIP.

Beyond RIP

The unique ability of RIP to convey routing information to a wide variety of hosts compels its use with the rank and file hosts. Its inadequacy for heterogeneous networks suggests that an ideal solution would use a more sophisticated protocol between routers and perhaps those servers attached to different sorts of networks, while retaining RIP for the low-end routing. With routers using RIP only for advertisements, and not for communicating with each other, the risk of routing loops that's usually associated with running multiple routing protocols is avoided. Translation of metrics from another protocol into RIP metrics is a problem, however.

The solution currently being pursued at SLAC is to use IGRP between the routers, with each router using RIP only to advertise a default route and routes to attached networks. The metrics of the default routes are tuned for each router to reflect the static distance from the firewall router, which attaches the SLAC network to the Internet. Multi-homed servers use RIP, with a bias against Ethernet routes which is explained below. A topic for future study is whether the servers should participate in the same routing protocol as the routers, which would force a move from IGRP to something else, probably OSPF or perhaps IS-IS.

The current solution still produces optimal routing locally, with packets from a host to a server on the same network going directly to the server and packets going to a network only one hop away going via the single-hop router. For destinations which are at least two hops away, a suboptimal route may be chosen in some circumstances. This does not seem to be too high a cost when set against the maintenance and reliability benefits of dynamic routing, and the route used is never worse than the routing produced by the static default route scheme which is being replaced.

While IGRP provides sufficient information to allow the routers to prefer FDDI routes, the multi-homed servers rely on inflation of RIP metrics to implement this preference. Host routes advertised to Ethernets are advertised with a inflated metric of 2 instead of 1, so routers or other servers which see both advertisements will choose the FDDI route because of the lower metric. In addition, servers add 1 to the metrics of all routes learned via Ethernets. This second addition or *biasing* is redundant for

metric	destination	via	note
1	server	FDDI	host route
2	server	Ethernet	server advertises higher metric
3	—	—	only seen within server
4	network	(any)	single hop to network
	default	(any)	from firewall router
5	—	—	only seen within server
6	default	(any)	from router on FDDI
7	default	Ethernet	from router **not** on FDDI

Figure 9a: Metrics as seen by host or router

metric	destination	via	description
1	server	FDDI	another server on FDDI
2	—	—	never seen by server
3	server	Ethernet	biased by receiving server
4	network	FDDI	single hop to network
	default	FDDI	from firewall router
5	network	Ethernet	single hop (biased by server)
	default	Ethernet	from firewall router (biased)
6	default	FDDI	from router on FDDI
7	default	Ethernet	from router on FDDI (biased)
8	default	Ethernet	from router **not** on FDDI (biased)

Figure 9b: Metrics as seen by server

Figure 9: Interpretations of RIP metrics

routes to other servers (which will end up with a metric of 3), but is required because the routers aren't flexible enough to inflate their own Ethernet advertisements.[3] (Figure 9 summarizes the interpretations of the metrics as seen from the differing viewpoints of servers and other devices.) With the addition of an FDDI connection, the first two sections of the **gated** configuration from Figure 5 are modified as illustrated in Figure 10.

With a metric of 3 representing a host route to a server via an Ethernet, routers use 4 as the metric for advertising routes to their attached networks. The firewall router also advertises a default route with a metric of 4. When fully implemented, the FDDI network will be the primary backbone of the SLAC network, and most Ethernets will only be a single router hop away from the FDDI, so routers attached to the FDDI advertise a default route with a metric of 6, metric 5 representing a (biased) Ethernet route to the firewall router. Other routers advertise a default route with even higher metrics. The relevant portions of the configuration for a typical router on the FDDI are shown in Figure 11.

This approach assumes that moving toward the FDDI network is good if the destination is not nearby, which may not always be the best choice. It also requires individual tuning of each router, though only in the metric for the default route and according to relatively straightforward rules. It thus is less general than one would wish for, though it seems to be approaching the best that can be accomplished within the myriad constraints, and without resorting to Byzantine configurations which aren't easily demonstrated to be correct.

[3]The routers don't listen to host routes, either, so the inflated metric of 2 applied by servers to host routes they advertise on Ethernets isn't yet used. The inflation is implemented on the servers now so the necessary information is available when host routes become supported by the routers.

```
# Enable RIP

rip yes {
    interface en0 metricin 2 metricout 1;    # bias against non-FDDI
    interface en1 metricin 2 metricout 1;    # bias against non-FDDI
};

# Static routes to our own interfaces -- handles for advertisements

static {
    host S_A gateway 127.0.0.1 noinstall;
    host S_B gateway 127.0.0.1 noinstall;
    host S_FDDI gateway 127.0.0.1 noinstall;
};
```

Figure 10: **gated** configuration modifications for server on FDDI

```
! configure RIP routing process (default is metric 6)
router rip
distribute-list 2 in
default-metric 3
network SU-SLAC
offset-list 1 out 3

! configure IGRP routing process
router igrp 4
network SU-SLAC

! metric inflation -- match all addresses
access-list 1 permit 0.0.0.0 255.255.255.255

! RIP redistribution -- only default route
access-list 2 permit 0.0.0.0
access-list 2 deny   0.0.0.0 255.255.255.255
```

Figure 11: Routing configuration for typical router on FDDI

Conclusions

Multi-homing of important servers, in concert with intelligent routing by hosts, provides a useful tool for improving performance and availability when client hosts exist on multiple networks. In an installation with hosts from many different vendors, the only protocol reasonable for distribution of routing information to the hosts is likely to be RIP. Despite limitations of both the protocol and the implementations, RIP can, with reasonable effort, provide optimal routing even in the presence of multi-homed servers so long as all networks use the same or comparable technologies. Introduction of a much faster or higher bandwidth network complicates the problem beyond the capabilities of RIP, though RIP can still convey essential routing information to hosts and allow them to route packets optimally to nearby destinations.

References

1. Stern, Hal. *Managing NFS and NIS.* O'Reilly & Associates, Inc., Sebastopol, California, 1991.
2. Pendry, Jan-Simon and Williams, Nick. *AMD – The 4.4 BSD Automounter Reference Manual.* Imperial College, London, March 1991.
3. *System and Network Administration*, pp. 382-384. Sun Microsystems, Mountain View, California, March 1990.
4. "The Network Information Service," in *System and Network Administration*, pp. 469-512. Sun Microsystems, Mountain View, California, March 1990.
5. Mockapetris, P. *Domain Names – Implementation and Specification (RFC 1035).* USC/Information Sciences Institute, Los Angeles, California, November 1987.
6. *MultiNet System Administrators' Guide*, p. 23-6. TGV, Inc., Santa Cruz, California, May 1991.
7. Hedrick, Charles. *Routing Information Protocol (RFC 1058).* Rutgers University, Rutgers, New Jersey, June 1988.
8. Information Technologies/Network Resources, Cornell University. "GateDaemon (Gated)" slides. Cornell University, Ithaca, New York, October 17, 1991.
9. *Router Products Configuration and Reference (vol. 2)*, pp. 14-71 – 14-73. Cisco Systems, Inc., Menlo Park, California, April 1992.
10. Deering, S. (ed.). *ICMP Router Discovery Messages (RFC 1256).* Xerox PARC, Palo Alto, California, September 1991.
11. *Shiva FastPath 5 Installation Manual*, p. 30. Shiva Corporation, Cambridge, Massachusetts, June 1991.
12. *PC/TCP Installation Guide*, p. 3-48. FTP Software, Inc., Wakefield, Massachusetts, January, 1991.

Acknowledgements

Many people provided a variety of assistance in this effort, including Mark Barnett, George Berg, Chuck Boeheim, Bob Cook, Les Cottrell, Krissie Griffiths, John Halperin, Fred Hooker, Tony Li, Jo Ann Malina, Greg Mushial, Don Pelton, Tim Streater, Mike Sullenberger, and to others whose contribution is not diminished by my failure to remember them. My gratitude goes to all of them. Thanks, too, to Alexander, for his patient tolerance, though I could have managed without the skunk.

Author Information

Karl Swartz is a member of the Systems and Networking Group within SLAC Computing Services at the Stanford Linear Accelerator Center, where he leads the UNIX support efforts. Prior to joining SLAC, he worked at the Los Alamos National Laboratory on computer security and nuclear materials accounting, and in Pittsburgh at Formtek, a start-up that is now a subsidiary of Lockheed, on vector and raster CAD systems. He attended the University of Oregon where he studied computer science and economics. Karl instructs at high-speed driving schools and enjoys good food and good beer (though not while driving) and long hikes with his Golden Retriever. Reach him electronically at kls@unixhub.slac.stanford.edu.

LADDIS: A Multi-Vendor and Vendor-Neutral SPEC NFS Benchmark

Andy Watson & Bruce Nelson – LADDIS Group & Auspex Systems

ABSTRACT

Distributed computing has been proven to be a cost-effective and productive model for many user environments. The benefits of this approach derive significantly from the ubiquitous availability of NFS (Network File System) licensed from Sun Microsystems, Incorporated [Sandberg85]. NFS is widely adopted as the de facto standard for shared online file storage, spanning the range from desktop PCs to supercomputers. An important decision in any large installation (such as those represented by participants at LISA) is the selection of one or more NFS file servers.

There are many ways to benchmark NFS file server performance and throughput. Comprehensive benchmark suites that exercise a target file server with parametrically tunable, scalable NFS client loads have historically been characterized by lack of reproducibility, or proprietary limits to availability/applicability, or instability at high loads, or tampering and misinterpretation. The LADDIS Group developed a new NFS file server benchmark suite based on significant enhancements to Legato's earlier NHFSstones.

This new benchmark (''LADDIS'') is well along in the process of adoption by SPEC as an official vendor-neutral industry-standard measure of NFS file server performance. In its current pre-release state, it may be licensed by any organization (subject to some restrictions on use of results obtained with it), under the name ''PRE-LADDIS.'' Ongoing testing and further development work (by the LADDIS Group and various SPEC member organizations) are refining LADDIS, with expectations of release late in 1992.

This paper describes LADDIS, with attention to its design intent, its underlying assumptions, usage advice, and interpretation of results. This is not an official document deriving from the SPEC organization; the authors are affiliated with the original LADDIS Group, related SPEC discussion groups, and Auspex Systems, Inc., a leading vendor in the NFS server marketplace. LADDIS overviews have appeared at previous conferences [LADDIS91] and in the press [Levitt91].

A Description of LADDIS

Overview, including History and Current Status

Over the past 30 months, engineers from Legato, Auspex, Data General, DEC, Interphase, and Sun (LADDIS) have met regularly to create the LADDIS NFS benchmark: an unbiased, standard, vendor-neutral, scalable NFS performance test. By the end of 1992, LADDIS should be a SPEC benchmark.

The purpose of the LADDIS benchmark is to (1) give users, system administrators, and network managers a credible and undisputed test of NFS performance, and (2) to give vendors a publishable standard performance measure that customers can use for load planning, system configuration, and equipment buying decisions.

The LADDIS Group, working cooperatively in a low-profile fashion and with high technical integrity, devised the LADDIS benchmark to eliminate user and vendor confusion about the plethora of previous, less-than-accurate, hard-to-run NFS benchmarks (e.g., the many versions of NHFSstones). As a result, LADDIS is a sophisticated Unix-based NFS traffic generator set of programs that measure file server performance in customer-sensible terms – throughput and response time.

Because of its accuracy and customer focus, the LADDIS NFS benchmark is being adopted by SPEC (the System Performance Evaluation Cooperative, creators of SPECmarks) as the first member of SPEC's System-level File Server (SFS) benchmark suite. LADDIS has been SPEC's first benchmark priority for many months, and has successfully passed several SPEC ''Benchathon'' sessions with multi-vendor LADDIS porting, testing, and revision. Many SPEC members such as HP, IBM, and Intel are now working with the LADDIS group to evolve the LADDIS benchmark into final SPEC form.

Explicit scalability is a major advantage of the LADDIS benchmark. This means that LADDIS can easily be used to fully capacity-test NFS servers with many Ethernet connections (e.g., 1-10), FDDI connections (e.g., 1-4), or disks (e.g., 1-100). It uses at least two Ethernet clients per server-attached Ethernet (more per server FDDI connection) to generate

NFS load on a target server. This NFS workload is distributed across file systems mounted on the available server disks. LADDIS causes this workload to grow linearly with NFS throughput, simulating the increased server network and disk activity of real-world NFS environments that are constantly adding new users.

Important but less-obvious LADDIS advantages are the benchmark's tunable control of subtle but crucial parameters such as: application workload; NFS operation mix; write-append ratio; file-size and file-count; and read/write-block-size distributions. The benchmark also specifies required settings for important external parameters like NFS write synchrony and UDP checksums.

An early version of LADDIS, called PRE-LADDIS, is currently available for beta test and evaluation from SPEC at no charge (see Appendix A). Until LADDIS is released by SPEC, **no** vendor's LADDIS performance figures can be publicly disclosed. The publicly available version of PRE-LADDIS shall be referenced as "version 0.1.0" elsewhere in this document.

The current internal version of PRE-LADDIS (non-distributed except to LADDIS Group and SPEC-SFS participants) contains some changes from the publicly available version 0.1.0 of PRE-LADDIS, mostly in the area of default values for various parameters. (At the time of this writing, the current internal version is 0.1.10.) These will affect the outcome of benchmark runs, in some cases significantly, even though the benchmark itself is fundamentally unchanged. For example, version 0.1.10 specifies that 70% of all writes to files will be appends, not overwrites, whereas the default for version 0.1.0 is only 5%. There are more reliable mechanisms for controlling larger numbers of load-generating clients in the current internal version, as well as a revised fileset size which scales linearly with the requested load.

Appendix documents provided with this paper are for the publicly available version 0.1.0 of PRE-LADDIS. All discussion of PRE-LADDIS herein shall be with reference to version 0.1.0 unless specifically cited otherwise.

Design Intent

There is no single model which encompasses all observed behavior of distributed systems using NFS for all application and network environments. However, based on research performed at Sun Microsystems, Inc., the assumptions built into LADDIS appear defensible specifically for CASE (Computer-Aided Software Engineering) environments, and also generally applicable to many other non-CASE sites. The parameter set which dominates the issue of validity for any given end-user environment is that of the mix of NFS operations generated by the benchmark.

LADDIS uses what has come to be known throughout the industry as the "standard mix" of NFS operations. These percentages, as shown below, are measurable with the (Unix) *nfsstat* command; if a different mix is desired, it can be specified easily (see Appendix B, the PRE-LADDIS man page).

null	getattr	setattr	root	lookup
0%	13%	1%	0%	34%
readlink	read	wrcache	write	create
8%	22%	0%	15%	2%
remove	rename	link	symlink	mkdir
2%	1%	0%	0%	0%
rmdir	readdir	fsstat		
0%	3%	1%		

For those interested in the history behind the derivation of the "standard mix", Bob Lyon of Legato Systems, Inc., has stated:

> The NFS operation mix [above] was taken from production servers inside Sun's operating systems group. The servers were 4/280s running SunOS 4.0. About 20% of the servers' clients were diskless, and the diskless servicing was done via NFS, not ND. Every diskless client was a Sun3, but the server had plenty of diskful Sun4s, Sun3s, and even a few Sun2s.

In addition to the standard mix, other key assumptions built into LADDIS involve the number, size, and distribution of files and the blocking used for I/O. Whereas Legato's now defunct NHFSstones benchmark had limited ability to tune these parameters, LADDIS makes tuning relatively easy (again, see Appendix B, the PRE-LADDIS man page). Furthermore, in order to better simulate real-world user environments, approximately 50% of all reads and writes are sized at 8 KB, with the remaining 50% a mix of 4-KB, 2-KB, and 1-KB fragments.

Perhaps the most appealing new feature of LADDIS is its ability to run multiple load-generating LADDIS clients from a single point of control, with all results being reported for individual clients as well as collated and summarized for the benchmark as a whole. The control role is performed by the LADDIS "Prime Client," which optionally can also be used to generate benchmark load.

Finally, to better model reasonable Ethernet loading behavior in the benchmark, SPEC Run Rules now under discussion include a requirement that a minimum of two clients per server network interface be used to generate load. This will protect against the artificial, unrealistic results which might be obtained with a single extremely fast client – a condition which is extremely atypical at installed user sites. This restriction applies only to results reportable officially via SPEC. As with all other LADDIS parameters, individuals are free to tune the

benchmark, running it under circumstances that best reflect their specific site requirements.

Known Limitations

Release 0.1.0 of PRE-LADDIS did not include any capability to model "asynchronous RPC" behavior such as that which occurs on typical client machines with several BIOD (Block I/O Daemon) processes. The current internal release includes the capability to imitate this behavior, and current SPEC Run Rules as drafted will require that a minimum amount of BIOD emulation will be required for all reportable results.

LADDIS lacks the ability to assign particular filesystems (called "mount points" or "MNT_PTS" in LADDIS documentation) to particular clients. All resources are assigned in a balanced round-robin manner. It would be interesting and useful in some instances to be able to direct some proportion of load-generating clients to exercise a particular system resource, while the others are equally distributed across remaining resources, but there is no support for this.

Of course, it is possible to run LADDIS in multiples of clients controlled by more than one Prime Client – or to run it on multiple individual clients with no Prime Client at all. In fact, some users of version 0.1.0 have reported on the exceptional utility of PRE-LADDIS as a load-generator during network debugging or profiling exercises. For such uses, it is more typical to start up individual LADDIS clients with multiple target servers distributed throughout a subnetted or otherwise-segmented network.

Another limitation of LADDIS involves neither the server being tested nor the benchmark software itself, but is an issue concerning the clients. Fast clients produce better (higher throughput with faster response times) LADDIS results than slow clients. To compensate for this, vendors are free to configure LADDIS to run the optimal number of LADDIS client processes per client machine (which affects context-switching overhead). This mitigates the problem but does not eliminate it. It is assumed that vendors will use fast clients to run LADDIS to obtain the best publishable results.

SPEC Run- and Report-Rules for Vendors

There has been some discussion, above, with respect to SPEC Run- and Report-Rules for vendors. At the time of this writing, it would be premature to divulge the full set of rules under discussion. It is expected that the full, official set of SPEC Run- and Report-Rules for vendors will be known in time to prepare additional materials for the LISA Conference in October, 1992. For now, discussion in this section of this paper is limited to a handful of representative rules which are very unlikely to change.

In addition to the requirements for a minimum of two load-generating clients per server network interface, and for a minimum amount of BIOD emulation (some number of read-aheads and write-behinds), the default values for many LADDIS parameters are being discussed. Please refer to the defaults as described in Appendix B and the documentation file, DESCR.LADDIS (included in the PRE-LADDIS distribution), but also be aware of the following changes.

As discussed in the overview, the percentage of writes which append to files instead of overwriting them has been changed from 5% in the publicly-available version 0.1.0 to 70% in version 0.1.10. This better corresponds with write operation activity as measured by Baker [Baker91, Hartman92].

UDP checksums will be required (the current default on some vendors' products is for UDP checksums to be disabled). This change is intended to reflect the need for this additional verification in environments that increasingly typically include intermediary devices (routers, gateways, etc.) where UDP checksumming is desirable in light of the greater probability of data corruption.

PRE-LADDIS 0.1.0 requires about 15 megabytes of disk space per client. An alternative approach under discussion within the LADDIS Group would have the fileset size scale linearly with the requested aggregate load, for example, at the rate of 3 megabytes per NFS IOPS (I/O Operation Per Second). Thus, a server being asked to deliver 1,000 IOPS would require 3 gigabytes of disk space to be exercised. The motivation for such fileset scaling is that larger client communities are typically served from correspondingly larger volumes of storage.

Vendors are also required to newfs (or equivalent) file systems immediately prior to testing. This is advisable for maximum reproducibility of results, and also to preclude any strategies for pre-heating file systems to improve performance artificially.

Still under discussion at the time of this writing are the results-reporting rules (format, admissible configurations, pricing information, whether to continue with "IOPS @ 50 ms" or use some other single figure of merit). It has been agreed that all LADDIS reports published by SPEC will include one or more Performance Plots (see figure 3.1), with multiple data points sufficient to characterize the tested server's performance and throughput under scaled load conditions. There is also a commitment to requiring compliance with the NFS version 2 protocol, which mandates synchronous writes to stable storage.

All configuration information for the server and clients must be reported, such that anyone attempting to reproduce the results could be successful.

Advice to Users of LADDIS

Considerations for Testbed Configuration

Uniform clients with generic configurations will work most easily to achieve consistent, meaningful results. Any attempt to run LADDIS using workstations in production use will suffer from the possibility that some clients are configured differently than others. The consumptive nature of LADDIS precludes simultaneous use of a load-generating client (or target file server) for any other production purpose during testing.

It is also very important to have known network conditions during the testing. It is best if the networks used for the testing are local to the server (no intermediate devices to affect results by adding latency or losing packets) and standalone (otherwise unused).

Spare disk space on the server is a must, of course.

Tunable Parameters and Labor-Saving Techniques

Users running LADDIS may find it interesting to tune the NFS operation mix in order to investigate extreme conditions. A good example of this is the use of 100% null operations, which will exercise the networks configured for the benchmarking without introducing any server-related variables.

Until it has been determined that everything is running properly, a shorter run time and warmup time can be used. The 600 seconds required as minimum Run Time in version 0.1.10 is conservative. Many servers give reproducible results with shorter run times. A little effort expended up front determining the optimal run time could cumulatively save hours in the long run.

```
PRE-LADDIS Single Client (hp4) Results, Fri Dec  6 12:10:41 1991
-----------------------------------------------------------------------

                                                Average   Average
NFS       Target Actual NFS    NFS      Cum.    Response  Response
Op        NFS    NFS    Op     Op       Time    Time      Time      Elapsed
Type      Mix    Mix    Count  Errors   (Sec)   (Msec/Op) Stddev    Time %
-----------------------------------------------------------------------
null        0%   0.0%       0       0     0.0       0.00      0.00     0.0%
getattr    13%  12.2%      67       0     0.4       6.74      5.33     1.7%
setattr     1%   0.7%       4       0     0.2      59.00     29.47     0.8%
root        0%   0.0%       0       0     0.0       0.00      0.00     0.0%
lookup     34%  35.1%     193       0     2.7      14.43     77.38    10.5%
readlink    8%   7.8%      43       0     4.5     105.09    225.59    17.0%
read       22%  23.8%     131       0     6.8      52.44     37.01    25.9%
wrcache     0%   0.0%       0       0     0.0       0.00      0.00     0.0%
write      15%  13.2%      73       0    10.3     141.56    239.49    39.0%
create      2%   1.2%       7       0     0.6      90.85     23.99     2.4%
remove      1%   1.6%       9       0     0.3      42.33     13.52     1.4%
rename      0%   0.0%       0       0     0.0       0.00      0.00     0.0%
link        0%   0.0%       0       0     0.0       0.00      0.00     0.0%
symlink     0%   0.0%       0       0     0.0       0.00      0.00     0.0%
mkdir       0%   0.0%       0       0     0.0       0.00      0.00     0.0%
rmdir       0%   0.0%       0       0     0.0       0.00      0.00     0.0%
readdir     3%   2.5%      14       0     0.1      14.07      0.91     0.7%
fsstat      1%   1.4%       8       0     0.0       5.87      2.17     0.1%
-----------------------------------------------------------------------

      ---------------------------------------------------------
      | PRE-LADDIS VERSION 0.00.16 SINGLE CLIENT RESULTS SUMMARY |
      ---------------------------------------------------------

NFS THROUGHPUT:    5.49 Ops/Sec   AVG. RESPONSE TIME:   48.19 Msec/Op
NFS MIXFILE: [ PRE-LADDIS Default ]
CLIENT REQUESTED LOAD: 5 Ops/Sec
TOTAL NFS OPERATIONS: 549         TEST TIME: 100 Sec
```

Table 1: Result Summary

The performance of the server and attached network(s) should be monitored. Knowledge of problems unrelated to LADDIS execution could spare much unnecessary frustration.

Initially, one may wish to disable some features so as to better understand their effects later. For future releases of LADDIS from SPEC, BIOD emulation can be disabled (there is no BIOD emulation in the publicly-available version 0.1.0). The write append ratio can be varied (keeping in mind that version 0.1.0 has a trivially small append ratio of 5%, and that increasing this to 70% shows what the server's disk subsystem can really do!), and the number of LADDIS processes per client (PROCS) should definitely be tested for optimal results.

Increasing the number of PROCS per LADDIS client typically increases the throughput (number of IOPS), but beyond a certain point (different for various models and makes of workstations) the law of diminishing returns takes effect. A lower value for PROCS improves performance (average response time), but may not produce the maximum number of IOPS. When in doubt, test three ways: with PROCS optimized for throughput, for performance, and for a moderate compromise of the two.

Interpretation of Results

LADDIS reports results for each individual load-generating client machine with statistics for enhanced interpretation. The Prime Client also collects results from all the individual clients and combines them into a consolidated overall average result. For each test trial, a data point is reported of the form "X IOPS @ Y Average-Response-Time". These points can then be plotted to show the scaling characteristics of the file server being benchmarked.

```
PRE-LADDIS NFS Benchmark Version 0.00.16, Creation - 5 December 1991
PRE-LADDIS Aggregate Results for 7 Client(s), Fri Dec  6 12:14:15 1991
--------------------------------------------------------
```

NFS Op Type	Target NFS Mix	Actual NFS Mix	NFS Op Count	NFS Op Errors	Average Elapsed Time (Sec)	Average Response Time (Msec/Op)	Elapsed Time %
null	0%	0.0%	0	0	0.0	0.00	0.0%
getattr	13%	11.9%	447	0	0.8	13.00	2.7%
setattr	1%	0.7%	29	0	0.3	96.71	1.2%
root	0%	0.0%	0	0	0.0	0.00	0.0%
lookup	34%	35.1%	1315	0	2.6	13.84	9.1%
readlink	8%	7.9%	298	0	2.9	68.70	10.3%
read	22%	23.6%	882	1	9.7	77.16	33.8%
wrcache	0%	0.0%	0	0	0.0	0.00	0.0%
write	15%	13.4%	502	0	9.9	139.43	35.0%
create	2%	1.5%	57	0	1.2	157.67	4.4%
remove	1%	1.5%	58	0	0.6	81.82	2.3%
rename	0%	0.0%	0	0	0.0	0.00	0.0%
link	0%	0.0%	0	0	0.0	0.00	0.0%
symlink	0%	0.0%	0	0	0.0	0.00	0.0%
mkdir	0%	0.0%	0	0	0.0	0.00	0.0%
rmdir	0%	0.0%	0	0	0.0	0.00	0.0%
readdir	3%	2.5%	97	0	0.1	13.11	0.6%
fsstat	1%	1.4%	54	0	0.0	5.40	0.1%

```
-------------------------------------------------------------------------
        ------------------------------------------------------
        | PRE-LADDIS VERSION 0.00.16 AGGREGATE RESULTS SUMMARY |
        ------------------------------------------------------
NFS THROUGHPUT:   37.40 Ops/Sec   AVG. RESPONSE TIME:   53.51 Msec/Op
NFS MIXFILE: [ PRE-LADDIS default ]
AGGREGATE REQUESTED LOAD: 35 Ops/Sec
TOTAL NFS OPERATIONS:   3740      TEST TIME: 100 Sec
NUMBER OF LADDIS CLIENTS: 7
```

Table 2: Another Result Summary

Examples of the text output of LADDIS appear on the following pages, in Tables 1 and 2 (extracted from the documentation file, DESCR.LADDIS, that accompanies the PRE-LADDIS distribution). An example of the Performance Plot appears in Figure 1.

At the time of this writing, SPEC has not reached consensus on the final format for these reports, but is expected to do so before October. The details of the final format for reported results will be provided during the presentation of this paper.

There is a wealth of information reported by LADDIS, most of which is only occasionally of significance. The exemplary aspect of this feature of the benchmark is that it allows the people running it to determine what information is significant!

It is always worthwhile to verify that the requested NFS mix is actually being achieved. When a file server or network is pushed to its extreme capabilities, the results are unpredictable, and one of the most obvious symptoms of this is a distorted operation mix. LADDIS does not re-attempt any operation which does not complete within its timeout window. Instead, these failures are reported as errors. Enough errors will certainly affect the mix.

It can also be useful to "reality-check" one's assumptions. Depending on server make, model, and configuration, one should have some theory as to its expected performance characteristics. For example, if a server with a disk susbsystem acknowledged to be slow is reporting 19-ms write operations, something is either very wrong or very strange (perhaps the filesystems were exported with the "async" option, or the SCSI disks are performing "immediate-reply" to write requests before the writes have completed to disk media?). If one adds very large amounts of memory for file caches on the server, one expects improved performance on all read-related operations. Verify the expected results by carefully examining the output reports.

The assessment of how well a file server performs is a function of its ability to sustain high loads without slowing down. The theoretical maximum number of standard-mix NFS IOPS per Ethernet is approximately 330. Taking into account a minimum of two LADDIS client machines per net, collisions will bring this down into the range of 300 IOPS. A file server that can handle 300 IOPS @ 50 ms might not be able to do so on a single Ethernet. Similarly, another file server capable of 300 IOPS @ 50 ms on one Ethernet might not be able to do 600 IOPS @

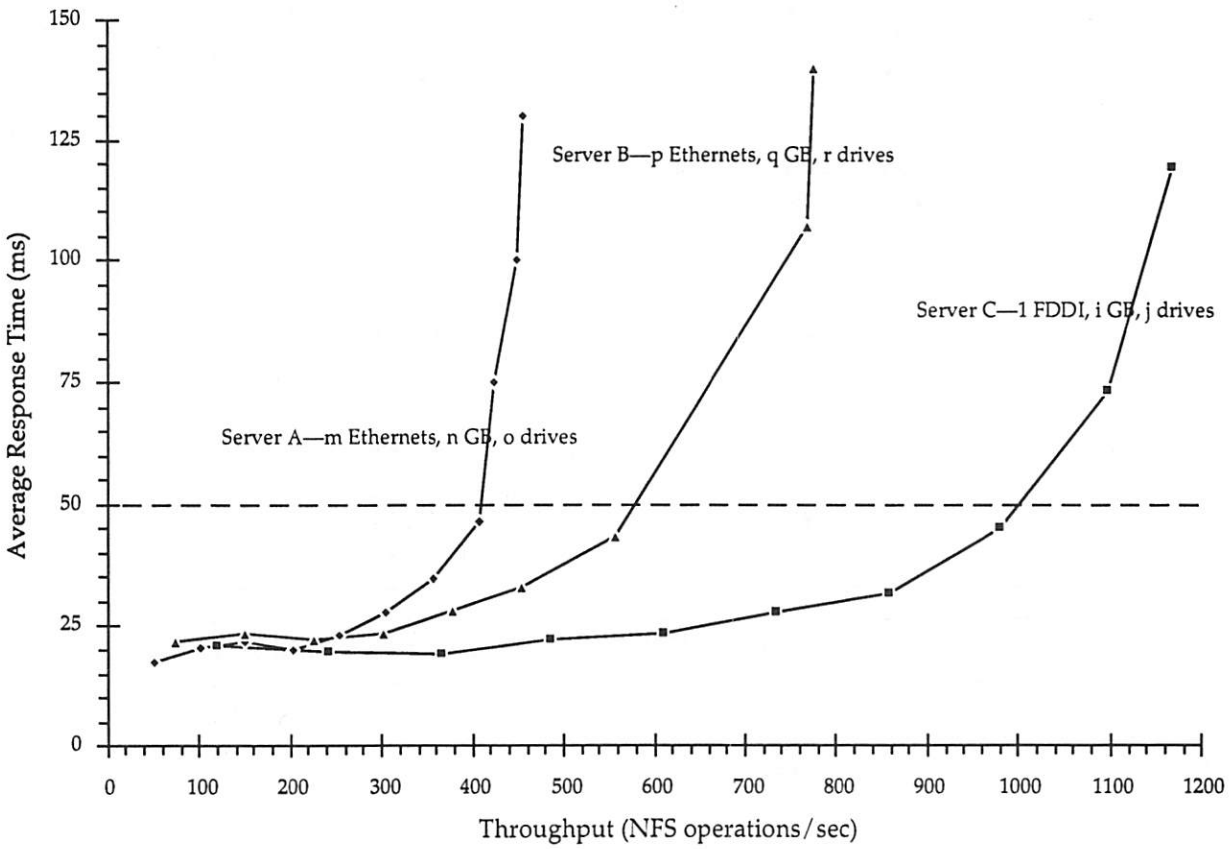


Figure 1

50 ms on two Ethernets, and so on, scaling upwards. Selection of a file server based on LADDIS results should take into account performance (average response time) and throughput (number of IOPS at that response time) for the site's *peak requirements*.

Table 1 shows an excerpt from the documentation file (DESCR.LADDIS) that is output typical of PRE-LADDIS 0.1.0. The actual values do not represent any particular server product.

A team of software developers working against a release deadline will take little comfort, waiting for software builds to complete, from the fact that the file server for their workgroup was chosen on the basis of the **average** load. Peak loads approach the theoretical maximum throughput capabilities of an Ethernet. It is important that the site's high-performance desktop workstations are not server-choked during critical phases of production activity.

LADDIS is an excellent tool for measuring NFS file server capabilities, but it will be best used under circumstances where the context for file server requirements are clearly understood.

Table 2 is an excerpt from the documentation file (DESCR.LADDIS) that is output typical of PRE-LADDIS 0.1.0. The actual values do not represent any particular server product.

References

[Baker91] Mary G. Baker, et al. *Measurements of a Distributed File System.* Proceedings of the 13th Symposium on Operating System Principles, October 1991.

[Hartman92] John H. Hartman. *File Append vs. Overwrite in a Sprite Cluster.* Sprite Project, University of California at Berkeley, Presentation to LADDIS Group on January 21, 1992.

[LADDIS91] The LADDIS Group. *LADDIS – A Vendor-Neutral Standard NFS Benchmark.* Proceedings of Interop 1991 Fall Conference, "Wednesday" volume, 9 October 1991. Free updated reprints are available from Auspex Systems, Inc.

[Levitt91] Jason Levitt. *Gauging NFS Server Performance [using LADDIS].* Unix Today, 25 November 1991, pp. 30 and 36.

[Sandberg85] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. *Design and Implementation of the Sun Network File System.* Proceedings of the Summer 1985 USENIX Conference, pp. 119-30, Portland, OR, June 1985.

Author Information

Bruce Nelson is chief technologist at Auspex. Between 1977-82, he was on the research staff of the Xerox Palo Alto Research Center (PARC), where he did pioneering work on distributed computer systems. He implemented the first and fastest remote procedure call (RPC) compilers (concepts now openly promulgated by Sun as ONC/RPC and OSF as DCE/RPC), developed and demonstrated practical optimization techniques for network protocol software, and built performance monitors for network file servers. Since PARC, Nelson has worked for both semiconductor and workstation companies, giving him a cross-spectrum understanding of computer systems issues. He joined Auspex in 1989, where he is responsible for product planning, benchmarking, and technology forecasting. He received his MS from Stanford University and his PhD from Carnegie-Mellon University, both degrees in computer science. Bruce is a member of Interop's Technical Advisory Board. Reach him electronically at BNelson@Auspex.com.

Andy Watson has a bachelor's degree in Natural Sciences and Mathematics (with a major in Physics) from Bard College, Annandale-on-Hudson, New York, 1979. Since 1980 he has been working as an end-user software developer, system administrator, network manager, consultant, and technical support engineer in various guises for many companies, including the New York City Transit Authority, Royal Digital Systems, Data General Corporation, Prime Computer, Inc., Martin Marietta Corporation, and (currently) Auspex Systems, Inc., where he is a member of technical staff. He has been representing Auspex at SPEC-SFS (LADDIS) subcommittee meetings. Auspex Systems is located at 2952 Bunker Hill Lane, Santa Clara, California 95054 USA. Phone: 408/492-0900. Fax: 408/492-0909. Reach him via electronic mail at AWatson@Auspex.com.

Appendix A: Formal Procedure for Requesting LADDIS code

The SPEC PRE-LADDIS beta test process became operational on Monday, February 3, 1992. This appendix describes the process as announced during the LADDIS Group's presentation at Uni-Forum '92 and also at Interop '91. The content of the beta test license and the license request process are consistent with the proposals approved by the SPEC Steering Committee at the January 1992 meeting in Milpitas, California.

The SPEC PRE-LADDIS beta test will consist of one beta test version of PRE-LADDIS distributed ONLY by electronic mail. The SPEC PRE-LADDIS Beta test software is licensed by SPEC, not by the LADDIS Group.

To obtain the PRE-LADDIS Beta test software, an individual must:

1. Request the SPEC PRE-LADDIS beta test License by electronic mail to "spec-preladdis-beta-test@riscee.pko.dec.com" with a subject line of "Request SPEC PRE-LADDIS Beta Test License".
2. Print a hardcopy of the license and sign.

3. Attach a cover letter written on the individual's company letterhead requesting the PRE-LADDIS Beta Test Kit.
4. U.S. Mail the signed license and cover letter to:

   SPEC PRE-LADDIS Beta Test
   c/o NCGA
   2722 Merrilee Drive, Suite 200
   Fairfax, VA 22031

After completing these steps, the SPEC PRE-LADDIS beta test kit will be emailed to the requestor from riscee.pko.dec.com. The licensee is requested to direct beta test comments via electronic mail to spec-preladdis-comments@riscee.pko.dec.com. This alias will forward all comments to the SPECSFS mailing list (which includes the LADDIS Group).

Note that PRE-LADDIS is ONLY available through electronic mail and ONLY through the process listed above in steps 1-4. If you do not have internet email available to you, you must arrange delivery of PRE-LADDIS through some email-capable part of your organization, not through LADDIS members like Auspex, DEC, Sun, etc.

Beta test kits should start shipping via email on February 18, 1992.

Note to magazine technical editors: As a consequence of SPEC's involvement in the LADDIS adoption process, magazines will NOT be allowed to publish PRE-LADDIS results. Thus, PRE-LADDIS may not be as useful as you expected it to be. However, you are still encouraged to obtain PRE-LADDIS and run in in your product evaluation labs, as your comments and suggestions in this beta-test period can enhance and improve the benchmark for your eventual, comparative-evaluation purposes. Certainly SPEC LADDIS (still scheduled for the end of 1992) will be useful for your product reviews, just as SPECmarks have been useful for CPU comparisons.

The LADDIS Group comprises:

Vendor	Name
Legato	Bob Lyon
Auspex	Bruce Nelson (LADDIS Chair)
Data General	Mark Wittle
Digital Equipment	Bruce Keith (SPEC SFS Release Manager)
Interphase	Vincent Lefebvre
Sun	John Corbin

NAME

PRE-LADDIS – Network File System server benchmark program

SYNOPSIS

prelad [–A append_pcnt] [–b blocksz_file]
[–B block_size] [–c calls] [–d debug_level]
[–D dir_cnt] [–e] [–F file_cnt] [–i] [–l load]
[–m mix_file] [–M prime_client_hostname]
[–N client_num] [–p processes] [–P]
[–S symlink_cnt] [–t time] [–T] [–V] [–z]
[–w warmup_time]
[directory ...]

DESCRIPTION

PRE-LADDIS is a Network File System server benchmark. It runs on one or more NFS client machines to generate an artificial load consisting of a particular mix of NFS operations on a particular set of files residing on the server being tested. The benchmark reports the average response time of the NFS requests in milliseconds for the requested target load. The response time is the dependent variable. Load can be generated for a specific amount of time, or for a specific number of NFS calls.

Normally, **PRE-LADDIS** is used as a benchmark program to measure the NFS performance of a particular server machine at different load levels. In this case, the preferred measurement is to make a series of benchmark runs, varying the load factor for each run in order to produce a performance/load curve. Each point in the curve represents the server's response time at a specific load.

PRE-LADDIS can also be used to characterize server performance. Nearly all of the major factors that influence NFS performance can be controlled using **PRE-LADDIS** command line arguments. Normally however, only the **–l load** option used. Other commonly used options include the **–t time** , **–p processes** , **–m mix_file** options. The remaining options are used to adjust specific parameters that affect NFS performance. If these options are used, the results produced will be non–standard, and thus, will not be comparable to tests run with other option settings.

Normally, **PRE-LADDIS** is executed via the **PRELAD_MGR** script which controls **PRE-LADDIS** execution on one or more NFS client systems using a single user interface.

PRE-LADDIS generates and transmits NFS packets over the network to the server directly from the benchmark program, without using the client's local NFS service. This reduces the effect of the client NFS implementation on results, and makes comparison of different servers more client-independent. However, not all client implementation effects have been eliminated. Since the benchmark does by-pass much of the client NFS implementation (including operating system level data caching and write behind), **PRE-LADDIS** can only be used to measure server performance.

Although **PRE-LADDIS** can be run between a single client and single server pair of machines, a more accurate measure of server performance is obtained by executing the benchmark on a number of clients simultaneously. Not only does this present a more realistic model of NFS usage, but also improves the chances that maximum performance is limited by a lack of resources on the server machine, rather than on a single client machine.

In order to facilitate running **PRE-LADDIS** on a number of clients simultaneously, an accompanying program called **PRELAD_MGR** provides a mechanism to run and synchronize the execution of multiple instances of **PRE-LADDIS** spread across multiple clients and multiple networks all generating load to the same NFS server. In general, **PRELAD_MGR** should be used to run both single- and multiple-client tests.

PRE-LADDIS employs a number of sub–processes, each with its own test directory named **./testdir<n>** (where <n> is a number from 0 to **processess** – 1.) If a test directory name already exists, then the **–e** flag specifies that the files already existing in the directory should be used for the test. Otherwise, a standard set of files is created in the test directory.

By creating symbolic links to existing empty directories on several mounted file systems, the load generated by each client can be spread over several server disks or disk partitions.

If multiple **directories** are specified on the command line, the **PRE-LADDIS** processes will be evenly distributed among the directories. This will produce a balanced load across each of the directories. If no directories are given, the current working directory is used as the test directory..

The mix of NFS operations generated can be set from a user defined mix file. The format of the file consists of the output from the **nfsstat(8c)** command (see the "–m" option below). The percentages taken from the mix file are calculated based on the number of NFS calls, not on the percentages printed by nfsstat. Operations with 0% in the mix will never be called by **PRE-LADDIS.** In a real server load mix, even though the percentage of call for a particular NFS operation may be zero, the number of calls is often nonzero. **PRE-LADDIS** assumes that the actual number of calls to these 0 percent operations will have an insignificant effect on server response.

PRE-LADDIS OPTIONS

–A append_pcnt
The percentage of write operations that append data to files rather than overwriting existing data. The append percent can be set from 0 to 100 percent. The default is 5% append.

–b blocksz_file
The name of a file containing a block transfer size distribution specification. The format of the file and the default values are discussed below under "Block Size Distribution".

–B block_size
The maximum number of Kilo-bytes (KB) contained in any one data transfer block. The valid range of values is from 1 to 8 KB. The default maximum is 8 KB per transfer.

–c calls The total number of NFS calls to generate during benchmark execution. The number of calls must be greater than the number of processes (see the "–p processes" option). The default is to execute the benchmark for a specific amount of time rather than generating a certain number of calls (see the "–t time" option).

–d debug_level
The debugging level, with higher values producing more debugging output. When the benchmark is executed with debugging enabled, the results are invalid. The debug level must be greater than zero.

–D dir_cnt
The number of directories to be used for directory operations. The directory count must be greater than 0. The default is 20 directories.

–e Use the existing set of files in the test directories for the benchmark. The default setting is to re-create a standard file set for the benchmark. The standard set of files is discussed below under "File Set". Note: This option may cause the benchmark to produce an invalid distribution of I/O packet transferred over the network.

–F file_cnt
The number of files to be used for read and write operations. The file count must be greater than 0. The default is 100 files.

–i Run the test in interactive mode, pausing for user input before starting the test. The default setting is to run the test automatically.

–l load The number of NFS calls per second to generate from each client machine. The load must be greater than the number of processes (see the "–p processes" option). The default is to generate 60 calls/sec on each/client.

–m mix_file
The name of a file containing the operation mix specification. The format of the file and the default values are discussed below under "Operation Mix".

–p processes
The number of processes used to generate load on each client machine. The number of processes must be greater than 0. The default is 7 processes per client.

–P Populate the test directories and then exit; don't run the test. This option can be used to examine the file set that the benchmark creates. The default is to populate the test directories and then execute the test automatically.

–S symlink_cnt
The number of symbolic links to be used for symlink operations. The symbolic link count must be greater than 0. The default is 20 symlinks.

–t time The number of seconds to run the benchmark. The number of seconds must be greater than 0. The default is 300 seconds. (Run times less than 300 seconds may produce invalid results.)

–T op_num
Test a particular NFS operation by executing it once. The valid range of operations is from 1 to 17. These values correspond to the procedure number for each operation type as defined in the ONC/NFS protocol specification. The default is to run the benchmark, with no preliminary operation testing.

–V Validate the correctness of the server's NFS implementation. The option verifies the correctness of NFS operations and data copies. The verification takes place immediately before executing the test, and does not affect the results reported by the test. The default is not to verify server NFS operation before beginning the test.

–z Generate raw data dump of the individual data points for the test run.

–w warmup
The number of seconds to generate load before starting the timed test run. The goal is to reach a steady state and eliminate any variable startup costs, before beginning the test. The warm up time must be greater than or equal to 0 seconds. The default is a 60 second warmup period.

MULTI-CLIENT OPTIONS

PRE-LADDIS also recognizes options that are only used when executing a multi-client test. These options are generated by **PRELAD_MGR** and should not be specified by an end-user.

–M prime_client_hostname
The hostname of the client machine where a multi-client test is being controlled from. This machine is designated as the "prime client". The prime client machine may also be executing the **PRE-LADDIS** load-generating code. There is no default value.

–N client_num
The client machine's unique identifier within a multi-client test, assigned by the **PRELAD_MGR** script. There is no default value.

OPERATION MIX

The **PRE-LADDIS** default mix of operations is:

null	getattr	setattr	root	lookup	readlink	read
0%	13%	1%	0%	34%	8%	22%
wrcache	write	create	remove	rename	link	symlink
0%	15%	2%	1%	0%	0%	0%
mkdir	rmdir	readdir	fsstat			
0%	0%	3%	1%			

A mix file can be created on a server by typing "nfsstat –s > mix". This command produces a count of each operation type along with its percentage. When a mix file is specified as a command line argument, **PRE-LADDIS** computes the mix percentages for each operation type based on the counts in the mix file rather than the percentages.

FILE SET

The default basic file set used by **PRE-LADDIS** consists of 100 regular files used for read and write operations, 20 directories used for directory operations, and 20 symbolic links used for symbolic link operations. In addition to these, a small number of regular files are created and used for non-I/O operations (eg, getattr), and a small number of regular, directory, and symlink files may be added to this total due to creation operations (eg, mkdir).

While these values can be controlled with command line options, some file set configurations may produce invalid results. If there are not enough files of a particular type, the specified mix of operations will not be achieved. Too many files of a particular type may produce thrashing effects on the server.

BLOCK SIZE DISTRIBUTION

The block transfer size distribution is specified by a table of values. The first column gives the percent of operations that will be included in a any particular specific block transfer. The second column gives the number of blocks units that will be transferred. Normally the block unit size is 8KB. The third column is a boolean specifying whether a trailing fragment block should be transferred. The fragment size for each transfer is a random multiple of 1 KB, up to the block size - 1 KB. Two tables are used, one for read operation and one for write operations. The following tables give the default distributions for the read and write operations.

Read - Default Block Transfer Size Distribution Table

percent	block count	fragment	resulting transfer (8KB block size)
0	0	0	0% 0 - 7 KB
85	1	0	85% 8 - 15 KB
8	2	1	8% 16 - 23 KB
4	4	1	4% 32 - 39 KB
2	8	1	2% 64 - 71 KB
1	16	1	1% 128 - 135 KB

Write - Default Block Transfer Size Distribution Table

percent	block count	fragment	resulting transfer (8KB block size)
49	0	1	49% 0 - 7 KB
36	1	1	36% 8 - 15 KB
8	2	1	8% 16 - 23 KB
4	4	1	4% 32 - 39 KB
2	8	1	2% 64 - 71 KB
1	16	1	1% 128 - 135 KB

A different distribution can be substituted by using the '-b' option. The format for the block size distribution file consists of the first three columns given above: percent, block count, and fragment. Read and write distribution tables are identified by the keywords "Read" and "Write". An example input file, using the default values, is given below:

```
Read
 0    0   0
85    1   0
 8    2   1
 4    4   1
 2    8   1
 1   16   1
Write
49    0   1
36    1   1
 8    2   1
 4    4   1
 2    8   1
 1   16   1
```

A second aspect of the benchmark controlled by the block transfer size distribution table is the network data packet size. The distribution tables define the relative proportion of full block packets to fragment packets. For instance, the default tables have been constructed to produce a specific distribution of ethernet packet sizes for i/o operations by controlling the amount of data in each packet. The write packets produced consist of 50% 8-KB packets, and 50% 1-7 KB packets. The read packets consist of 90% 8-KB packets, and 10% 1-7 KB packets. These figures are determined by multiplying the percentage of type of transfer times the number of blocks and fragments generated, and adding the totals. These computations are performed below for the default block size distribution tables:

Read percent	blocks	fragments	total blocks	total fragments
0	0	0	0	0
85	1	0	85	0
8	2	1	16	8
4	4	1	16	4
2	8	1	16	2
1	16	1	16	1

----	-----
149	15
90%	10%

Write percent	blocks	fragments	total blocks	total fragments
49	0	1	0	49
36	1	1	36	36
8	2	1	16	8
4	4	1	16	4
2	8	1	16	2
1	16	1	16	1
			----	------
			100	100
			50%	50%

USING LADDIS

As with all benchmarks, **PRE-LADDIS** can only provide numbers that are useful if the test runs are set up carefully. Since it measures server performance, the client (or clients) should not limit throughput. The goal is to determine how well the server performs. Most tests involving a single client will be limited by the client's ability to generate load, not by the server's ability to handle more load. Whether this is the case can be determined by running the benchmark at successively greater load levels and finding the "knee of the curve" at which load level the response time begins to increase rapidly. Having found the knee of the curve, measurements of CPU utilization, disk i/o rates, and network utilization levels should be made in order to determine whether the performance bottleneck is due to the client or server.

For the results reported by **PRE-LADDIS** to be meaningful, the tests should be run on an isolated network, and both the client and server should be as quiescent as possible during tests.

High error rates on either the client or server can also cause delays due to retransmissions of lost or damaged packets. **netstat(8)** can be used to measure the network error and collision rates on the client and server. Also **PRE-LADDIS** reports the number of timed-out RPC calls that occur during the test as bad calls. If the number of bad calls is too great, or the specified mix of operations is not achieved, **PRE-LADDIS** reports that the test run is "Invalid". In this case, the reported results should be examined to determine the cause of the errors.

To best simulate the effects of NFS clients on the server, the test directories should be set up so that they are on at least two disk partitions exported by the server. NFS operations tend to randomize disk access, so putting all of the **PRE-LADDIS** test directories on a single partition will not show realistic results.

On all tests it is a good idea to run the tests repeatedly and compare results. If the difference between runs is large, the run time of the test should be increased until the variance in milliseconds per call is acceptably small. If increasing the length of time does not help, there may be something wrong with the experimental setup.

The numbers generated by **PRE-LADDIS** are only useful for comparison if the test setup on the client machine is the same across different server configurations. Changing the **processes** or **mix** parameters will produce numbers that can not be meaningfully compared. Changing the number of generator

processes may affect the measured response time due to context switching or other delays on the client machine, while changing the mix of NFS operations will change the whole nature of the experiment. Other changes to the client configuration may also effect the comparability of results.

To do a comparison of different server configurations, first set up the client test directory and do **PRE-LADDIS** runs at different loads to be sure that the variability is reasonably low. Second, run **PRE-LADDIS** at different loads of interest and save the results. Third, change the server configuration (for example, add more memory, replace a disk controller, etc.). Finally, run the same **PRE-LADDIS** loads again and compare the results.

SEE ALSO

The **DESCR.LADDIS** SPEC benchmark description file contains many pointers to other files which provide information concerning LADDIS.

ERROR MESSAGES

illegal calls value
The **calls** argument following the **–c** flag on the command line is not a positive number.

illegal load value
The **load** argument following the **–l** flag on the command line is not a positive number.

illegal procs value
The **processes** argument following the **–p** flag on the command line is not a positive number.

illegal time value
The **time** argument following the **–t** flag on the command line is not a positive number.

bad mix file
The **mix** file argument following the **–m** flag on the command line could not be accessed.

can't find current directory
The parent process couldn't find the pathname of the target directory. This usually indicates a permission problem.

can't fork The parent couldn't fork the child processes. This usually results from lack of resources, such as memory or swap space.

can't open log file
can't stat log
can't truncate log
can't write sync file
can't write log
can't read log
A problem occurred during the creation, truncation, reading or writing of the synchronization log file. The parent process creates the log file in /tmp and uses it to synchronize and communicate with its children.

can't open test directory
can't create test directory
can't cd to test directory
wrong permissions on test dir
can't stat testfile
wrong permissions on testfile
can't create rename file

can't create subdir
A child process had problems creating or checking the contents of its test directory. This is usually due to a permission problem (for example the test directory was created by a different user) or a full file system.

bad mix format: unexpected EOF after 'nfs:'
bad mix format: can't find 'calls' value
bad mix format: unexpected EOF after 'calls'
bad mix format: can't find %d op values
bad mix format: unexpected EOF
A problem occurred while parsing the **mix** file. The expected format of the file is the same as the output of the **nfsstat(8)** command when run with the "–s" option.

op failed: One of the internal pseudo–NFS operations failed. The name of the operation, e.g. read, write, lookup, will be printed along with an indication of the nature of the failure.

select failed
The select system call returned an unexpected error.

BUGS

PRE-LADDIS can not be run on non–NFS file systems.

Shell scripts that execute **PRE-LADDIS** must catch and ignore SIGUSR1, SIGUSR2, and SIGALRM, (see signal(3)). These signals are used to synchronize the test processes. If the signal one of these signals is not caught, the shell that is running the script will be killed.

FILES

./testdir*
per process test directory
/tmp/prelad%d
process synchronization log file

NFS Performance And Network Loading

Hal L. Stern & Brian L. Wong – Sun Microsystems Computer Corporation

ABSTRACT

Current synthetic NFS benchmarks provide a fair view of peak server performance but do not measure network loading and its impact on perceived server response time. We performed several experiments with many NFS clients, using both token ring and Ethernet networks, in an attempt to quantify NFS network loading with both diskless and dataless clients. Our benchmarks consisted of Constant Client Load (CCL) tests, designed to measure the impact of adding a new client to an existing network, and Constant Network Load (CNL) tests to determine how the number of NFS clients generating a given load affects the network. The benchmark results demonstrated that large numbers of clients cause significant increases in server and network latency, even when the total NFS load generated by all clients is held constant. We use the results of these experiments to comment on the utility of collision rates for measuring network load and to compare NFS scalability to that of other distributed filesystems.

Introduction

Multiple client benchmarks provide fair measurements of expected NFS server performance in a computing environment modeled by a particular workload "mix." These synthetic benchmarks are usually run with few clients, however, while most actual networks have many more machines on them. The effects of multiple clients may keep the performance of an NFS server below its theoretical limit, due to a variety of effects:

- Clients use a larger working set of files on the server. This leads to more disk seek operations (since more of each disk is in use), and possibly unbalanced disk activity. Increasing the number of files in use also places a greater load on the directory name lookup cache (DNLC) and inode cache on the server.
- Increasing contention for the network. If the same workload is distributed over more machines while keeping the total load on the network constant, then each node on the network experiences an increase in network latency.

Our goals were to quantify the effects of incremental network growth on NFS performance and to determine valid measurements of network loading and network latency under a variety of client loads. We then discuss the actual benchmark configuration and tests, results and short analyses of each experiment followed by conclusions and ideas for future work. Finally, we present graphs of benchmark results.

Benchmark Process

Two basic types of experiments were used:

- *Constant Client Load* benchmarks (CCL), in which the load generated by each client was held constant as clients were added to the network. These experiments measured the effects of incremental network growth in an attempt to answer questions like "How will adding two more workstations affect the network?"
- *Constant Network Load* benchmarks (CNL) fixed the total load on a network and distributed it evenly across N clients. Variable load benchmark results show how a fully loaded NFS server will behave with a varying number of client machines.

Each of the benchmarks was run on Ethernet and 16 Mbit/sec token ring networks, using each of the NFS load mixes described below.

Client and Server Configurations

The server used for these experiments was a Sparcstation 2 running SunOS 4.1.1, with 16M of memory, an SBus Prestoserve accelerator, and four 669M SCSI disks on two SCSI host adaptors. Client data was spread evenly over four disks, with two disks on each host adaptor. Two internal 207M SCSI disks were used for the root, swap and /usr filesystems on the server. Server system tuning included setting maxusers to 64 and running 16 nfsd daemons.

The network clients were Sparcstation 1s with 16M of memory, also running SunOS 4.1.1. Each client ran a version of the *nhfsstone 2.0.3*. NFS benchmarking utility modified to retry `mount` RPC calls that timed out. This change allowed many clients to participate in a benchmark, even if the server could not immediately handle all of their initial mount requests. To further reduce the initial burst effects, nhfsstone clients staggered their mount requests by a up to 10 seconds. Running the benchmarks over a 10-minute period made the several second delays insignificant in calculating total client load.

Each nhfsstone client used four processes instead of the default seven processes. The default nhfsstone configuration is useful for saturating a network with only a few clients, which was contrary to the goals of these experiments. Reducing the number of clients processes reduced the working set of files used by each client. However, this client-side workload generation closely matches the behavior of a single user sitting at a workstation, running a window system, a mail application, and one or more interactive terminal sessions. In each CNL benchmark, the total NFS client load was chosen to be about 80% of the maximum capacity of the NFS server, preventing the server itself from becoming a bottleneck.

NFS Models

The nature of an NFS workload is colored by the hardware configuration and usage of the network clients. Diskless clients tend to be write-heavy, since they must continually write swap pages back to their boot server. Dataless clients (those with local disks for root and swap) generate fewer NFS write requests, since most write operations are directed to the local disk; they tend to be weighted more heavily in read, getattr and lookup calls. Adding memory to client machines also tilts the mixture in favor of getattrs. Increasing the effective size of the client's NFS page cache has a corresponding increase in the number of cache consistency checks done via getattr requests.

Our benchmarks used three NFS RPC mixtures: the default Legato mix, emulating a diskless client; the Brown University mix, representing a dataless client; and a hand-crafted mixture modeling a client that requests read-mostly access. This last RPC mixture is based on characteristics of clients that handle large data files, and is called the "Large" mix. The Brown (dataless) client mixtures substitute getattr operations for the write calls generated by diskless clients. The large number of files used by the nhfsstone benchmark provides a good model for a software development environment, a classroom-student lab, or an automated office. The files used in these configurations are typically less than 100 Kbytes each. In the Large RPC mixture, the read operation weighting is increased at the expense of the number of lookup RPCs generated. This is characteristic of a workstation that reads very large files: name-to-file handle translations are done once and reading those files' pages dominates the NFS workload. Image processing, VLSI layout and verification, and earth resources applications routinely use files that are several hundred megabytes in size, and typify the Large client.

The RPC call compositions of the three workloads are shown in Table 1.

The Legato RPC mixture, dominated by write operations, maximizes the number of hosts that will be transmitting on the network at the same time. The dataless client mixtures are heavy in operations bound by server responses - getattr, read and lookup - and tend to generate a more "well-ordered" traffic pattern during the benchmark. In this sense, "well-ordered" means that there is less randomness among the writers on the network. When client write operations dominate the RPC mixture, there are many nodes on the network that are transmitting bursts of packets at the same time. When read operations compose the bulk of the NFS workload, the NFS server is the primary transmitter of multi-packet trains and the client nodes send only small requests to the server.

Operation	Legato	Brown	Large
getattr	13	23	22
setattr	1	1	1
lookup	34	45	30
readlink	8	8	5
read	22	13	33
write	15	2	1
create	2	1	1
remove	1	1	1
readdir	3	5	5
fsstat	1	1	1

Table 1: RPC Call Compositions

Going from a write-intensive to a read-intensive configuration changes the load on the network from one in which many clients are generating bursts to one in which only one (or a few NFS servers) are sending bursts of packets. Both CNL and CCL benchmarks were run using each of the three NFS client RPC mixes.

Analysis of Results

We review the benchmark results by first comparing Ethernet and token ring results from the CCL experiments, and then make some generalizations about network and server loading based on the CNL benchmarks.

Incremental Network Loading

The CCL benchmark data presented in and Figures 1 & 2 suggest that 16 Mbit/sec token ring and Ethernet networks perform comparably under increasing client loads. Using the Legato RPC mixture, Table 2 shows that Ethernet and token ring reach their peak acceptable performance ranges at about the same throughput and response times.

	Clients	NFSops	Resp. Time
Token ring	14	20	45 msec
Ethernet	20	14	58 msec

Table 2: Peak Performance Ranges

With the Brown mixture, all 24 clients were placed on the network without exceeding 70 msec response time, typically taken as an acceptable upper limit on

server response time. The Large RPC mixture showed similar results, although the 24 client Ethernet case raised the server latency to 74 msec.

Figures 1 and 2 show that both Ethernet and token ring networks experience non-linear increases in network latency after some number of clients have been added. Although 20 clients is the threshold on Ethernet while the token ring limit is 14 clients, each Ethernet client is generating less of a load. To gain an exact measurement of the number of clients at the "knee" of the performance curve. both Ethernet and token ring clients should have generated identical loads; this was a deficiency in the experiment. The total number of NFSops/sec generated by all clients is about the same in both cases (280 NFSops/sec). There does not appear to be any appreciable difference in the incremental growth behavior of token ring or Ethernet networks using several realistic NFS workloads.

On a token ring network, the maximum network latency is fixed, independent of the number of frames sent on the network. Token ring latency is always the time required for the token to circulate around all N nodes of the network. If each node on the network has the same latency, then the maximum latency is simply Nt, where t is the latency for one node. The token ring latency increases linearly as nodes are added, provided each node produces a workload similar to other nodes on the network. In reality, each node will either immediately forward the token or transmit a token and a packet, with widely varying latencies. In a moderately loaded network, the average latency for all nodes can be considered equal: some nodes will not be transmitting and will immediately forward the token, while others attach a frame to the token. Under moderate to high load conditions, the average latency for each node can be considered to be the same, since the ring will have an essentially constant total traffic load with each node contributing equally to the total.

On Ethernet, the expected latency is theoretically unbounded, but has a realistic bound at some large value. In general, the CSMA/CD mechanism creates a non-linear relationship between the latency per node and the number of nodes. Under low loads, Ethernet has a low latency, but under high loads, the latency increases faster than that of token ring. A lightly loaded Ethernet will perform better than a lightly loaded token ring, because of the fixed overhead of the token passing mechanism. The actual crossover point at which the expected token ring latency is equal to that of Ethernet depends greatly on network load and the number of clients. In the "middle ground" around this point, Ethernet and token ring are comparable, as the data produced by these network loading benchmarks has indicated. Beyond the crossover point, token ring offers lower average network latency than Ethernet, but the volume of traffic produced by the clients in this configuration may be more than the NFS server(s) can handle. Quite simply, when enough nodes are added to a token ring network so that its average latency is lower than that of an Ethernet, the network itself may have become a major bottleneck in NFS performance.

DNLC Bottlenecks

Some of the non-linear increase in server response time is due to thrashing of the directory name lookup cache (DNLC) on the server. Each nhfsstone client has a working set of about 200 files; adding clients increases the total number of file name to inode mappings that the server must manage. With more than 16 clients on the network, the DNLC hit rate dropped below 85%, and with 24 clients on the network the DNLC hit rate was as low as 55% on the NFS server. A possible solution is to hand-tune the DNLC size, without blindly increasing the *maxusers* parameter, to prevent growing other kernel data structures such as the process and system file tables.

The flavor of work performed on NFS clients determines the probability of running into DNLC thrashing. In a software engineering environment, header files, libraries, and documentation are frequently shared by many developers. Due to sharing, the working set of files in this case is *not* simply the number of files used by each client multiplied by the number of clients. *nhfsstone*, however, models this coarser working set count. In an office automation environment, it is more likely that each workstation user will have a more independent group of files in use: users have their own copies of mail, reports, spreadsheets and "filing cabinets" that are merged only occasionally. Increasing the cardinality of the server's working set of files may create a bottleneck at the DNLC. That is, even if the total set of files isn't growing in aggregate disk space used, an increase in the number of files used may adversely affect NFS performance.

Are Collision Rates Useful?

One of the goals of the CNL benchmarks was to measure the effects of spreading the same load over more network clients. The CNL experiments showed a slight performance gain using token ring instead of Ethernet networks when the network clients are the primary generators of traffic bursts on the network. In the Legato benchmark cases, Ethernet performance began to degrade when only six clients shared the network traffic load; while on token ring up to ten clients could be generating traffic before server response time began to approach 70 msec. Refer to Figures 3 & 4 for a graphical comparison.

Using the Brown and Large client models, in which the server generates most long packet trains (in response to read requests), both token ring and Ethernet networks show a flat response time curve,

until 16-20 clients are running on the network. The kind of workload generated by the clients determines the relative impact of network loading. Furthermore, this data suggests that there is no "set rule" for placing M clients on a single ethernet. If some or all of the clients are dataless workstations then inter-client contention will be of minimal impact.

In the CNL experiments on Ethernet, the response time for the Legato run rises sharply as the same load is spread over more clients, but the collision rate drops after an initial peak. (Refer to Figures 5 & 6). One explanation is that the additional latency is caused by deferred transmissions, rather than transmissions that resulted in collisions. With a few clients generating more packets per second, it was more likely that two clients would begin to transmit within the same time window. When the constant network load is distributed across more clients, each station generates less traffic, and the carrier sense mechanism is able to avoid collisions. The total network latency is not decreased, however, because time lost to retransmitted packets after collisions is more than exceeded by the time wasted deferring transmission while a carrier is present on the Ethernet.

In the dataless client model experiments, the constant client load (on Ethernet) produced a relatively constant number of collisions as clients were added to the network. Again, the NFS server response time increases sharply with large numbers of clients, but network loading characteristics remain somewhat constant as the number of clients varies. In the dataless client RPC mixtures, there are very few write operations, so the NFS server is the only host sending long, 8kbyte datagrams on the network. NFS clients are primarily receivers of data, as the result of read operations, than the transmitters of data through write NFS calls. With this "better ordering" of the network traffic, clients see less network latency.

The CNL data for Ethernet benchmarks shows that the collision rate itself is not a reliable indicator of network loading; the actual mix of operations and distribution of load across all NFS clients should be considered as well. With more "talkers" on the network, the probability of a high deferred transmission rate on each client is increased. To determine if a network is overloaded, both collision rate and the packet defer rate should be considered.

Scalability of NFS

Measuring effects of incremental loading and distributed workload should leads us to comment on the scalability of NFS. From the CCL benchmarks, in which similar clients were added to the network until the server's peak performance was reached, it is obvious that scaling the size of an NFS network usually produces a decrease in performance seen by the NFS clients. Even with an appropriately large, well-configured server, additional load on the network and the server increases response times.

To keep NFS performance relatively constant while increasing the client count, it would appear necessary to offset the larger number of clients with a decrease in the work done by each client. Keeping the network load constant was the goal of the CNL benchmarks, which also showed a decrease in perceived client performance as the workload was distributed across larger numbers of machines.

One goal, perhaps, is to keep an NFS server away from the high end of its capacity. This model is similar to operating a transistor in its active region rather than always driving it to full-on or full-off states; in the active region the behavior of the device can be more finely controlled. Corroborating this idea is the physical fact that no single network or set of networks can be scaled infinitely. Capacities of several thousand NFS operations per second available from high-end NFS servers, but this does not imply that doubling that capacity will provide better NFS performance.

Gaining any real measure of scalability from NFS clients implies finding ways to make them generate fewer operations:

- Add memory to the clients to increase the size of their NFS page caches.
- Improve the NFS cache consistency algorithm to eliminate cache consistency checks for read-only or "durable" files that are rarely changed.
- Cache recently used files on disk, doing cache fills and flushes to the NFS server in large segments and only when required.

When comparing the scalability of different distributed filesystems, it is critical to compare similarly configured systems. An Andrew File Sysstem (AFS) client, for example, uses local disk caching to reduce traffic between client and server. Compare AFS with dataless clients, such as those modeled by the Brown RPC mixture, and you'll find much better NFS scalability than if you compare AFS and diskless clients.

Conclusions and Future Work

Variations in server and network loading under the three different client RPC mixtures highlights the need for additional characterizations of NFS performance. Several studies of local and distributed filesystems have suggested that most UNIX files are small and relatively short-lived, while acknowledging that some large files may be used by the types of clients we have classified as using the "Large" model. Most NFS benchmarking and performance measurement tools focus on the workload presented by the clients and the server's response times without tracking the file reference patterns of the clients. One possible metric to develop is a time-

weighted average of file sizes used by NFS clients: The size of each file read (or written) in its entirety would be given greater weight in the average the longer the client used that file. The goal of this instrumentation is to determine how client memory configurations would affect NFS performance: small files used for long periods of time or several large files would both benefit from increased client memories.

As with all benchmarks, results must be discounted by the differences between the benchmark and the proposed end user environment. Differences in network and server loading between the Legato, Brown and Large client mixtures demonstrate that client behavior and the number of clients are determinants of user-perceived response time.

Benchmark Data

In the graphs of client response times, the gray reference lines bracket the 40-70 msec range of acceptable server response times.

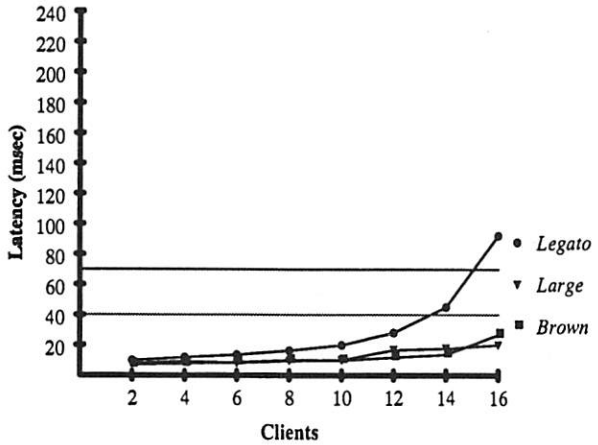


Figure 1: NFS Response Time vs. Number of Token Ring Clients – Constant Client Load

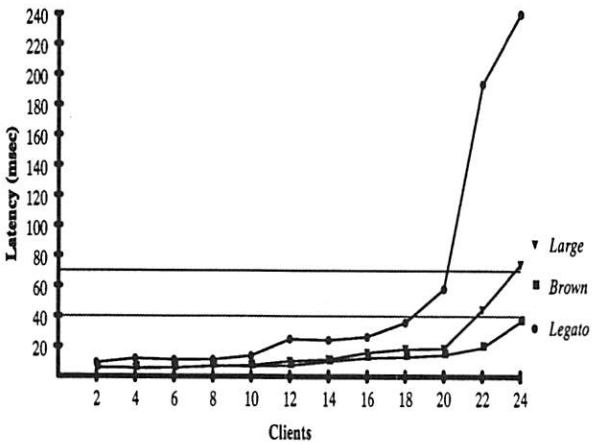


Figure 2: NFS Response Time vs. Number of Ethernet Clients – Constant Client Load

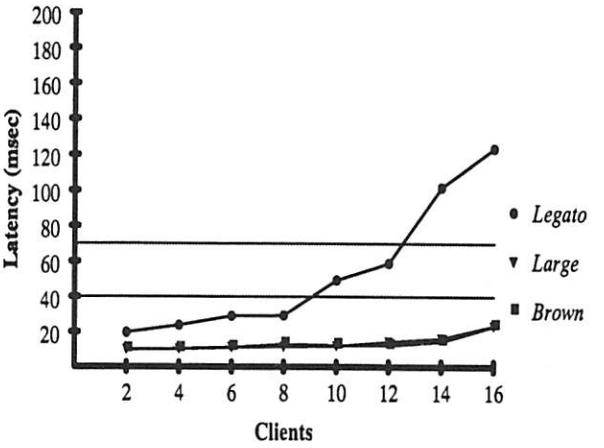


Figure 3: NFS Response Time vs. Number of Token Ring Clients – Constant network Load

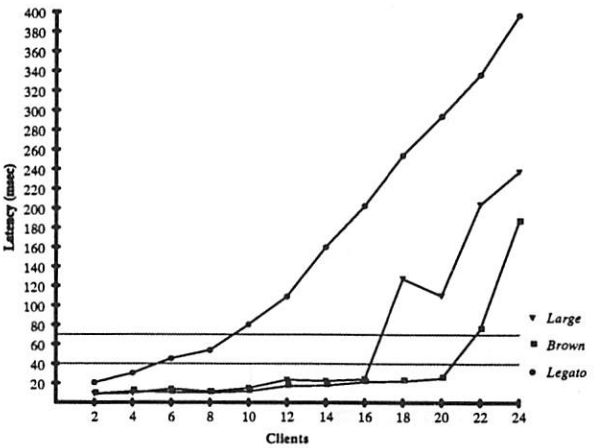


Figure 4: NFS Response Time vs. Number of Ethernet Clients – Constant Network Load

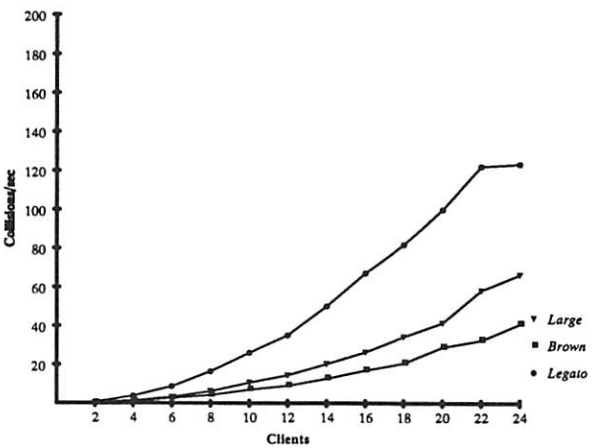


Figure 5: Collision Rate vs. Number of Ethernet Clients – Constant Client Load of 14 NFSops/sec per client

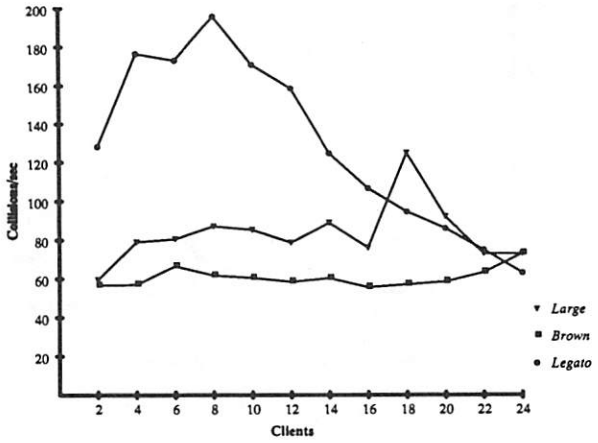


Figure 6: Collision Rate vs. Number of Ethernet Clients – Constant Network Load of 360 NFSops/sec

Acknowledgements

We would like to thank Nahn Chu, Rajiv Khemani and Varun Mehta for the use of their performance laboratory and their valuable feedback. Peter Galvin of Brown University has been a constant source of ideas, empirical data and good humor.

References

[Brown92] Technical Staff, "A Summary of Network Performance Benchmarking of Sun 4/490 File Servers", Computer Science Department, Brown University, December 1990.

[Keith90] Bruce Keith, "Perspectives on NFS File Server Performance Characterization", in Proc. USENIX Summer Conference, pp. 267-277, Anaheim, CA, June 1990.

[Satya92] Mahadev Satyanarayanan, "The Influence of Scale on Distributed File System Design", IEEE Transactions on Software Engineering, Vol 18, No 1, January 1992.

[Baker91] Mary Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout, "Measurements of a Distributed File System", Proc. of the 13th ACM Symposium on Operating Systems Principles.

[Boggs88] D.R. Boggs, J.C. Mogul and C.A. Kent, "Measured Capacity of an Ethernet: Myths and Reality", Proc. ACM Sigcomm '88, Computer Communication Review, Vol 18, No. 4, August 1988, pp.222-234.

Author Information

Hal Stern holds a BSE from Princeton University. He has worked for Intermetrics, Inc, Princeton, and Polygen Corporation in capabilities ranging from compiler bug fixer to system administrator and product designer. Hal joined Sun Microsystems in 1989 and is currently an Area Systems Engineer. US Mail should be addressed to Sun Microsystems, 55 Old Bedford Road, Lincoln, MA 01773, with electronic mail to hal.stern@east.sun.com.

Brian Wong ran his own consulting business and worked for British Telecom before joining Sun in 1987. Brian was a Systems Engineer in the Federal Sales area and joined the Corporate Technical Marketing staff last year. US Mail should be addressed to Sun Microsystems, 2550 Garcia Drive, MS PAL1-431, Mountain View, CA 94043, with electronic mail to brian.wong@corp.sun.com.

Dropping the Mainframe Without Crushing the Users: Mainframe to Distributed UNIX in Nine Months

Peter Van Epp & Bill Baines – Simon Fraser University

ABSTRACT

This paper describes the first phase of the evolution of the computing environment at Simon Fraser University from a centralized, long established, mainframe based computing environment to a distributed UNIX based computing environment.

The first section of the paper gives a brief history of the site and some of the previous initiatives towards distributed computing.

The second section describes the Task Force report that mandated a conversion to UNIX within 9 months and the plan that was developed to do that.

The third section details the migration plan versus migration reality.

We then extract from this experience the major issues encountered in converting from a mainframe environment to UNIX. We also mention some things that are taken for granted in the mainframe world that don't seem to exist in the UNIX world. This should be of interest to sites planning to convert from a mainframe to UNIX, and to UNIX vendors that are interested in selling machines in that market. We also describe how we resolved some of the problem areas.

Next we describe what we did right, what we did wrong, and what we would repeat if given the opportunity. We also describe some UNIX experiences encountered when pushing the 'shrink wrap' envelope with an implementation that started out with 18,000 user accounts.

The last sections of the paper detail some of the new work we are undertaking to resolve existing problems, and add additional function. Our conclusions attempt to summarize our major points.

History of the Site

Since the mid 1970s central academic computing at SFU was done on a series of IBM mainframes running the Michigan Terminal System (MTS). MTS was initiated at the University of Michigan and was enhanced and maintained by UM and the 7 other universities that run it. Since MTS was written specifically for the academic time sharing environment it evolved over the years into a very secure system, (as the students found security holes they were plugged) with a very rich set of access control mechanisms. There are however, several problems with the model. With no manufacturer supporting the operating system, each site was required to have staff capable of doing operating system kernel level development and support. Staff at this level are both hard to find and hard to keep, and since there are so few sites using the system, training them was a large burden. The mainframe that MTS runs on is expensive in both capital cost and maintenance. There are many things that a mainframe does very well (high volume, high speed I/O comes to mind), but equally many things that a mainframe does poorly (interactive graphics support, character by character I/O).

In the early 1980s, the MTS community recognized that the computing world was changing and that distributed computing was the wave of the future. There were discussions about the activities that the various sites would undertake to move to a distributed environment and what role, if any, MTS would play in that environment. One thing that became apparent was that the "dumb" terminal over telephone wire to the central machine was going to give way to a network of machines distributed across many different areas. As a result of this realization, and concurrent with a major telephone system upgrade at SFU, the computing department designed and installed the physical portion of a new high speed network. At that time (1985), it was not clear whether Ethernet or Token Ring would ultimately dominate, and FDDI was on the horizon. The decision was made to create two major nodes each connected to the other by multi-mode fibre optic cable that would hopefully support FDDI when it became available. These two network centers were connected to wiring closets (initially 38 and now up to 50 and climbing) via multi mode fibre. The wiring closets were connected via shielded twisted pair

cable to every place where there was a phone installed. The cable could support either token ring or Ethernet connections with equipment available at that time, and now of course, supports 10BaseT connections and looks like it will be able to support twisted pair FDDI when that becomes available.

With this infrastructure in place we then proceeded (as funding permitted) to connect the various Macs and PCs around campus into departmental Lans (using the new high speed network wiring) and allow them to access the mainframe, (through a new PDP11 based interface that supported TCP/IP and TN3270 into the IBM mainframe and therefore MTS) and the Internet, either directly or via logging on to MTS and then going out to the Internet.

As the start of the phaseout of MTS, we identified specific functions that could be better done in a distributed manner and started implementing them with the vision that at some point in the future, there would be no more user functions on MTS. MTS would either operate as a "server" for various functions, or be replaced by some better and cheaper system – with minimum user impact. The first function to be replaced was text processing. The norm at that time was an nroff like package (called Textform) that ran on MTS and the standard MTS text editor. Both of these were clearly inferior to the packages available on the Macs and PCs, and we therefore installed 75 IBM PS2 model 25's and 75 Mac Plus's with Microsoft Word, one dot matrix draft printer for each 3 machines and 2 LaserWriters for final output. This facility was in the University library, and was named 'The WordStation'. It quickly became the place of choice for all types of word processing. Support for Textform text processing was removed from MTS.

The next easily identifiable group of users was the Numerically Intensive Computing users. At that time they were using all the cycles that they could get on the mainframe, but since they were sharing it with up to 200 time share users, they obtained fewer CPU cycles than they wanted and they impacted response for the other users. Their jobs were CPU bound, did almost no I/O and therefore they didn't benefit from the I/O performance of the mainframe. There were only 8 to 10 users in this category, and they were moved to a pair of 4 processor Silicon Graphics (SGI-240) UNIX machines. Today, this SGI complex has 20 processors across three systems and is still growing. At the time of the switch over, there were reports that the users were seeing as much as an 8 times increase in performance on their jobs. They were no longer competing with the 200 time share users, the SGI CPUs were each more powerful than the IBM mainframe in CPU performance.

These two steps reduced the load on the mainframe enough that a new machine did not have to be purchased. The next step was to size up an instructional microcomputer facility. A questionnaire was sent to all faculty asking what kind of microcomputers, what software and how many hours per week they needed in support of courses that could use computing. The response prompted us to purchase a lab of 75 IBM PS2 386sx PCs. These machines were connected to the high speed network to allow connections to the mainframe and the Internet, and today, are supported by three Novell servers for file services and software. In addition, a reservation program running on Novell was written that allows an instructor to book a certain number of hours per week per student for a course. The software then allows the student to book a machine in advance for a specific time period to do his or her course work, at times when the machine is not booked in this way it may be used on a 'walk in' basis for course work. This facility reduced the load on the mainframe even further. Today it is overbooked by almost 200% and still is not used to its full capacity (i.e. the faculty are over estimating the amount of computer time the students need, some students have their own PCs and some students drop out).

This was basically the state of the world at SFU on Feb. 6, 1991 when the report from the task force on computing was released, and computing at SFU was once again drastically changed.

The Task Force Report

In August of 1990, the computing center stopped reporting to the Vice President for Research and started reporting to the Associate Vice President Academic. The new Vice President commissioned a task force of senior academics to examine and make recommendations on computing at SFU. On Feb. 6, 1991, the task force released its report, and the Associate Vice President acted upon the recommendations.

The recommendations included:

1. UNIX is the academic operating system of choice, and SFU should move to it.
2. By August 31, 1991, the electronic mail system will be moved to a platform other than MTS.
3. By December 31, 1991, all computing will be moved to UNIX and MTS will be shut down.
4. The IBM 3081 mainframe should be decommissioned on Dec. 31, 1991.
5. The direction of computing at SFU will be in the hands of a hierarchy of committees composed of representatives from the user community, and these committees will set computing policy and direction for the computing center.
6. The computing center will be split into two new departments,
   6a) Academic Computing Services – responsible for the implementation and

support of all academic computing.

6b) Operations and Technical Support – responsible for the day to day operation of all the central site machines, both academic and administrative.

The staff of the computing center that had been supporting the administrative systems were distributed to the various administrative departments, and those departments became responsible for the maintenance and support of their own applications.

The staff of the operations group was not significantly impacted.

The recommended time table implied that we had to convert from a mainframe environment with very little UNIX experience to a production UNIX environment within 9 months. UNIX support in the computing center at that time consisted of a single UNIX consultant that we had hired away from our School of Computing Science. He was supporting a departmental Sun 4/280 as a pilot UNIX service for both the department and the community, and the Numerical Computing SGI complex. Very few other people in the department of 60 people that remained after the split up had ever even used UNIX and none of those except for the lone UNIX consultant had ever done any serious UNIX system administration.

The Plan

Once the shock had died down a little (one consequence of the committees taking over the policy making role was 6 of the computing center managers were made redundant and no longer had jobs), we began planning how we were going to get from where we where to UNIX, and how "UNIX" would look. For the first several months, there were meetings pretty much every day: first to identify MTS functionality, and then to target replacement services in the brave new world of UNIX.

The committees proposed by the task force report were commissioned, and they started to meet regularly to discuss how the changes would be made. Some committees solicited input from the user community.

One of the first decisions made was that the new UNIX system would be a distributed system, and therefore there would not be a mainframe-like UNIX host, but rather a series of smaller UNIX hosts. Given the shortness of the time frames involved, it was further decided that the conversion would be done in two stages. The first stage would be to replace exactly the functionality of the current mainframe system with some number of UNIX systems that would reside in the central machine room (this to be done by the deadlines specified in the task force report) followed by whatever further distribution the community saw fit once some experience with UNIX was gained by both the users and the computing staff.

The gathering of the research community's application requirements and the list of replacement applications for existing MTS services was distilled down to a list of what packages were available on UNIX (with costs) and a list of applications that might have to be dropped. From this, a manageable list of what we proposed to support was formulated. The committees' initial cut on a configuration was to buy a server for each of the 37 departments at SFU, but when the cost of software (and the fact that the software costs were per machine) was compared to the budget, it was quickly decided that 5 servers, one for each faculty, was more in line with what could be afforded. Several of the schools realized that their software requirements were similar, so they merged their servers. The final plan included three research servers, divided along the lines of function; a database server, a statistics server, and a compute server. Access to these machines would be restricted to faculty and graduate students. The instructional committee chose two machines for instructional use, and a general login server for e-mail and conferencing was specified. The initial implementation total would be six hosts (3 research, 2 instructional, and 1 general purpose).

The committees were surprised at the true cost of distributed computing. During their early consultations they discovered that the cost of multiple copies of software across many UNIX hosts could easily exceed the prices being paid for mainframe software. Some software vendors even classed new RISC UNIX machines as supercomputers, and charged appropriately. The reduced initial capital costs of the UNIX hardware did not fully offset the surprisingly high software costs.

In parallel with the committee effort, we were profiling the current usage of MTS, this necessitated making modifications to MTS to collect the data. The data collection ran for a few months with the following results:

E-mail and Conferencing	28%
Statistical packages	20%
Compilers	15%
Custom Applications	18%
Symbolic Mathematics	4%
Numerical Analysis	4%
Text Processing	4%
Utilities	2%
Databases	3%
Graphics	2%

We identified every function we could find that was being used on MTS, and summarized the results in a document. We then identified adequate UNIX based replacements, and those that had less than adequate UNIX based replacements. A few functions had to be dropped.

At the same time, another group of the computing center staff were examining the issues of providing a UNIX service and deciding on a recommended architecture for the UNIX system to be submitted to the committees for approval. The two tasks, detailing what we had now, and deciding on what the UNIX system should look like, of course interact to a large extent, with one influencing the other, and so there was a lot of intergroup discussion.

The broad outlines of what we thought the UNIX service should look like follows:

- All people that are members of the university community should have automatic access to a UNIX account for free. This was largely true of the MTS system, with undergrad students being eligible for a resource limited account on MTS on request.
- The model of MIT's Project Athena for access should be followed if possible (i.e., the Athena motto "wherever you go there you are") so that the users home directory and environment should be the same no matter what machine he or she is logged on to.
- In support of the last statement about access, and to continue the mainframe concept of backup of user files on a regular basis at an affordable cost, there should be a single central file server that holds all of the users files and can serve them over the network to whatever machine the user is logged in to.

This all having been worked out, we then turned our attention to how all of this would be implemented. This was going to be a major and disruptive change for everyone involved especially the researchers who had been using MTS for many many years and had programs written to take advantage of packages and features on MTS that wouldn't be available on UNIX. In order to give them time to move, it was desirable to have the UNIX system up by July 31, so they would have all of summer semester to move their code over to UNIX. Starting with the fall semester (starting in September) all student computing would be moved off of MTS (if possible) leaving us with a fallback to MTS for the fall semester if problems occurred, allowing an orderly shutdown in December.

The Reality

As always the plan failed to survive its first brush with reality. We had thought that we were treading down a well traveled road from a mainframe to UNIX. We discovered that this isn't true. We were not able to find a site (nor were the various UNIX workstation vendors) running large numbers of users on one or more server class machines. It is possible that the exclusion of the mainframe like UNIX vendors was the cause of this, if you have a mainframe, and are moving to UNIX maybe you choose a large UNIX box and move to that.

Restricted Main Memory Bandwidth

The major challenge in moving from a mainframe to UNIX servers appears to be providing sufficient I/O bus and main memory bandwidth on the UNIX server at a cost that doesn't approach that of a mainframe. The UNIX vendors keep pumping up the CPU power of their boxes, but have not been matching the CPU power with an equivalent increase in main memory bandwidth, possibly because doing so isn't cheap.

In order to understand why this is so, a little bit of background on main memory is required. The typical workstation uses Dynamic Random Access Memory (DRAM) as their main memory. DRAMs have typically been doubling in size every couple of years while falling in cost, making them an inexpensive main memory. The access speed of DRAMs, however has not been falling at any where near the rate of the size increase. Due to packaging constraints (pins on the package) and chip architecture (buffers are expensive in chip real estate) the trend has been toward ever larger numbers of bits in a single package (1 megabyte SIMMs, 4 megabyte SIMMs, soon 16 Megabyte SIMMs). This has a rather large performance impact, since typically only 1 byte out of those x million bytes in the SIMM can be accessed at a time, and the cycle time to get to the next one can be several hundred nanoseconds. Note that the cycle time of a DRAM is different (and longer) than the often quoted access time. A DRAM with an access time of 70 nanoseconds has a cycle time of up to several hundred nano seconds.

If we assume that the typical SIMM can be cycled in 200 nanoseconds, then the SIMM has a memory bandwidth of 5 megabytes per second. If we assume a 4 byte wide memory bus, then this gives us a total main memory bandwidth of 20 megabytes per second. For comparison, a typical mainframe has a main memory bandwidth that is between 200 and 800 megabytes per second, obtained by using static ram in place of the DRAM, and using memory interleaving to obtain even more speed (but at vastly higher cost).

The next thing to consider is that this 20 Megabyte per second bandwidth into memory has to be shared between the RISC CPU, and the I/O subsystem that is transferring data to and from the various peripherals. The RISC CPU executes (on average) one 32 bit instruction per clock cycle and clock speeds of 25 to 50 Megahertz are not uncommon. If not for instruction and data caches, this would consume all of the available main memory bandwidth feeding the CPU, leaving no main memory bandwidth for doing I/O. If we assume that the caches are such that half the main memory bandwidth is available for I/O, we are left with around 10 megabytes per second to service the I/O requirements of all of the peripherals.

Restricted I/O Paths to Peripherals

Now let's look at the I/O subsystem of a typical IBM mainframe, as noted above, a fancy (and expensive!) main memory subsystem provides an order of magnitude higher main memory bandwidth (even if the cost is probably 2 orders of magnitude larger in the mainframe case). This leaves a considerable amount of bandwidth for the use of the I/O subsystem. The I/O subsystem consists of its own processor (typically called a channel director) that controls the transfer of bytes to or from main memory and the peripherals. The channel director has between 16 and 256 channels (on old to modern mainframes). Each channel can transfer data to a peripheral at 3 megabytes per second, and the main memory bandwidth is such that many (but perhaps not all) of these channels can be transferring data at the same time without causing instruction starvation for the CPU which is also contending for main memory.

Think of an IBM channel as being similar to a SCSI controller on a UNIX box. To replace our mainframe which had 4 channels out to disk, we would need a UNIX box with 4 SCSI controllers each controlling a string of disks, and all capable of being active at the same time. In addition, there were another 3 channels out to high performance tape drives that are not present on the UNIX machines and that I/O load has now moved to the disks as well.

If all 4 of those channels transfer data at the same time, we would need a path from the disks into main memory on the UNIX box of 12 megabytes per second, but the whole main memory bandwidth is only 20 megabytes per second, and there are still the other peripherals (such as the Ethernet interface) that need some of that I/O bandwidth. Our smallish mainframe is already taxing the I/O capacity of a single UNIX box. As you can see, a larger mainframe site with more channels to disk would quickly overwhelm a single UNIX box.

The obvious answer to this problem is to spread the I/O over multiple UNIX boxes (which is what we did). The challenge becomes to do this in such a way that no single path in the system becomes a choke point to the I/O flow.

Establishing a Distributed File System

There are two major choices for a central UNIX file server, NFS or AFS. We chose NFS not because we think it is better, but because it is already supported by most UNIX vendors, and our single UNIX person had experience with NFS but not with AFS. Given how little UNIX experience we had, the custom nature of AFS, and the tight time frames, We believe this was the correct choice. NFS expertise can be purchased since there is a lot of NFS around; we expect that AFS expertise is harder to find. That said, AFS is certainly a viable solution, and may well be where we end up in a few years. The advantage to AFS is the reduction in load on the backbone network due to local caching. In our current configuration this is not a problem, if we move towards placing workstations on people's desks supported by the central file server (e.g., to provide cost effective backup services) then this will become an issue. One of our fellow MTS sites, Rensselaer Polytechnic Institute (RPI) has chosen to go with AFS to support their cluster of some 400 UNIX workstation seats (some are X terminals), and they have successfully implemented it. The decision will probably hinge on how distributed a site is initially going to be, what networking is in place and how much UNIX expertise is in place.

That decided, we started talking to the various file server vendors about I/O performance on NFS servers. At that time, we found there were few choices (several vendors had just released products, but had neither performance numbers nor a track record), and there little understanding of the issues involved (with the notable exception of Auspex).

The issue here is server performance. We had the choice of buying several (probably 4, one for each disk channel) NFS servers, and then cross mounting them all onto the server machines, or buying a single large server and allowing it to serve all the machines with no cross mounts. If we were to use the single large server, then we are right back to the main memory bandwidth and I/O performance limits of a UNIX box. In order to get the necessary bandwidth to disk, we had already decided that each server machine would have 2 Ethernet cards, one to connect to the backbone and accept telnet connections from users and one dedicated to NFS traffic. This meant that the server had to be able to support between 4 and 6 Ethernet ports and (at this point) 10 gigabytes of disk all fighting for that same main memory bandwidth. In addition the CPU would have to be fielding all of the interrupts and doing the Ethernet and NFS processing for all this I/O. The Auspex solution of dedicating a CPU to each function (one CPU to each two Ethernet ports, 2 disks per SCSI controller all tied together by a very high speed bus) made more sense to us than any alternative we saw.

We ordered an Auspex NS5000 NFS file server with 10 gigabytes of disk and 6 Ethernets to spread the I/O load around to the 6 server machines. We ordered 4 Silicon Graphics 4D320 machines (2 processor 33 mhz R3000 CPUs) for the three research machines and the general login server, and 2 Sun 470s for the instructional machines. As noted above, each of these machines has a second Ethernet card to connect to one of the 6 Ethernet ports on the Auspex file server to provide NFS service in a way that is both secure (because of the private net) and does not add load to the campus backbone.

Analyzing the Computing Load

The choices of machines were arrived at like this: around 30% of the MTS load of 180 simultaneous logins was taken up by E-mail and conferencing, NetNews access (which didn't exist on MTS) was expected to add some more load, so the general login server was allocated 80 out of those 180 users (40%), but they would all be light duty. The statistical users were another 20% and were basically divided evenly between two of the research machines at 20 users each the third research machine was set to be also 20 users. The remaining 40 users were assumed to be students and split 20 users each to the two instructional machines. The remaining 20 people were considered taken up by the RS6000/530 that takes intermediate CPU users (defined as needing between 500 and 1000 CPU hours per year, as opposed to large scale users whose CPU demand is essentially infinite!).

Three Sun Sparc2 machines provide infrastructure machines, one for NIS, NetNews, and syslog; one for DNS, X.500, and the mail hub, and the third as a PostScript converter and driver for our Xerox 4090 laser printer. A VAX/VMS system was reinstalled to run several services that seemed to be easier to implement under VMS.

The 180 user number used above was arrived at because that was the typical load on the MTS machine at the end, and the initial commitment was to provide exactly the same level of service on the UNIX hosts, no more, no less. This was to be true because the UNIX conversion was being funded with the same money currently supporting the mainframe (which as you can imagine caused interesting problems during the time when both sets of machines existed and were being paid for with the same money ...), and any enhancements had to be approved by the committees with the indication of where funding would be found to pay for the enhancement.

Unfortunately, when these machines were ordered, the earliest delivery we could get was mid August – there went the "convert the researchers over the summer" theory, the machines would not arrive in time.

At about this point, we realized that in order for E-mail to be cut over about 90% of the work of the conversion was going to have to be done (i.e., being able to get mail on the UNIX box implied that you be able to log on to the UNIX box, which in turn implies that you have an account and a home directory on the file server etc.) This started a process of talking both the Registrar's Office and the Personnel Department out of the data for students (from the registrar) and faculty and staff (from personnel) so that we could create accounts for everybody. The file server and the SGI CPUs arrived the same day (Aug 26) and both were up within a couple of days (fine support from both companies!) and the mad race to get the systems configured and some 20,000 accounts and home directories set up and installed was underway.

In actual fact, the E-mail cutover happened on Sept 12 (12 days late) after a huge effort (including one 72 hour straight session for two of the people babysitting the account / home directory creation task). This turned out to be too late for the fall semester student computing load too, so it went ahead on MTS. The next several months saw additional software packages and any number of problems found and fixed, and in fact MTS was shut off to user access on January 2, 1992 right on schedule. Delay in the installation of terminal servers to replace the PDP11s meant that MTS actually was still up until March 13, 1992. This turned out well, since many people didn't believe MTS would actually go, and therefore hadn't moved any of their files. MTS being up allowed us to FTP the files rather than attempt to recover them from tape. All in all, this conversion went far more smoothly (and in a far more timely fashion) than anyone believed possible.

Major Issues Raised by the Conversion from a Mainframe to UNIX

Many issues confronted us in the conversion from a long entrenched mainframe system to UNIX, the next section will discuss how they were resolved.

- I/O bandwidth: the IBM 3081 mainframe has 16 I/O channels, each in theory capable of transferring data at 3 megabytes per second and enough main memory bandwidth to allow most of those 16 channels to be transferring data at full speed. Monitoring tools in MTS indicated that the MTS system was as often I/O bound as it was CPU bound with the normal 180 or so users on the system. There is no particular reason to believe that the same would not be true on a UNIX machine supporting the same workload.
- Politics indicated that we couldn't replace the mainframe with a mainframe like UNIX system, indicating the I/O rate had to be distributed across a number of smaller machines (and therefore be somewhat balanced!).
- Magnetic tape support. The mainframe supports both 6250 bpi 12 inch "round" tape drives (IBM 3420s) and IBM 3480 "square" cartridge tapes, with a high bandwidth path (two IBM channels) into the machine. It is very common for a user with a large data set to mount the tape and process the data on the mainframe directly off the tape without copying it to disk first. MTS also can read both all types of IBM standard labeled tapes, ANSI labeled tapes and unlabeled tapes.
- The high speed tape drives make system

backup a not too time consuming job. In the MTS case (as in the safe UNIX case), the system is brought down to single user mode for weekly full backup. This involves about 3 hours to back up some 15 gigabytes of disk to 3480-type tapes.

- Tape library services: MTS provided support for access control of labeled tapes. The label of the tape contained both a tape serial number and the MTS id of the tape's owner. When a tape is mounted, the MTS tape software checked the MTS account name mounting the tape against an access control list before allowing the mount. This is common on mainframe systems, as is having a tape library system to control access to the tapes. At 10,000 tapes we have a smallish tape library, libraries of 100,000 tapes are not uncommon in commercial shops.
- The PDP11 front end system provides an interface to both the SFU library computer and the public switches X.25 service (Datapac in Canada). Since there is a hard dollar cost associated with this, the service needs to be accounted for by user.
- High speed printing: There is a Xerox 4090 92 page per minute laser printer attached to MTS and driven from an IBM channel for campus printing needs. It was printing around 600,000 pages a month when MTS was in use (split between MTS printing and printing from the administrative systems).
- Accounting: the MTS system accounts for CPU time, disk space usage Virtual Memory usage while the job runs, network connect time, printer usage, and also provides limits on all of these resources by user account.
- As pointed out earlier, MTS is a very secure system, offering access protection by access control lists down to the level of a single account (or exclusion down to a single account) for read, write or various kinds of execution (i.e. it is possible to restrict what programs a user can execute against a data file by user id). All of these protections support full wild carding of parameters.
- User directory: MTS implements a directory of all users on the system, which can be searched (via a soundex algorithm) to find a persons E-mail address from an approximation of their name.
- E-mail features: the E-mail system on MTS provides a rich variety of archiving and filtering mechanisms for incoming E-mail.
- Conferencing: a very popular custom conferencing system (called forum) had been written on MTS. A replacement system needed to be found for UNIX.
- A replacement for the PDP11 based custom terminal interface to the mainframe needed to be found.
- Education: both the computer center staff and the user community need to be trained in UNIX.
- In order to create the necessary 18,000-20,000 UNIX accounts that would be required to give everyone on campus a UNIX account, an automated method of account creation would have to be found. There were around 11,000 active accounts (i.e., ones that had accessed files within the last year on MTS, at the point of the switchover).

These issues were resolved in many different ways:

- All users' home directories (currently some 25,000 accounts, some 11,000 of which are active) and any data needed on more than one machine, is stored on an Auspex NS5000 NFS file server. As noted above, this machine has 6 Ethernet ports and started out with 10 gigabytes of disk, and is now up to 18 gigabytes. One Ethernet port connects to the campus backbone, and 4 others are used for dedicated connections to the server machines for both security and load sharing reasons. The last Ethernet port is used over a dedicated fibre link to the undergraduate computing lab comprised of 30 NeXT stations and 5 or so Sun servers of various types. This allows the Computing Science students access to their campus home directory from the Computing Science lab as well as our server machines.

  A public lab of 40 NeXT stations will be added to one of the other Ethernet ports (along with a router) in the near future.

  The Auspex fileserver exports only the undergraduate partitions to the Computing Science lab, and the same will be true of the NeXT lab. None of the Auspex partitions are exported to the campus backbone. The reason for this is the potential insecurity of the NFS protocol. If a machine where a user can become root has access to the Ethernet carrying the NFS data, then that user can, without much difficulty, read any open file whether or not he should have access. We avoid this problem by a combination of things: not exporting important file systems (i.e.; those of faculty, grad students, and staff) to undergraduate labs. We also export the file systems to our server machines on a second Ethernet port and Ethernet that runs between the Auspex and the various servers that are all in a physically secure machine room. This means that an attacker has to either break root on one of our UNIX hosts, or to tap one of the private Ethernets in order to tap the NFS data.
- As noted earlier, there are 6 Server machines split by function and each with a dedicated Ethernet port for NFS traffic. In addition,

there is an RS6000 model 530 that is used for a group of 20 or so medium scale researchers, and 3 Sun SparcStation2s. One Sparc2 does NIS, NetNews, syslog, and terminal server security support. Another Sparc2 supports the DNS, the X.500 server, and is the campus mailhub. Each of these two machines have three Ethernet ports providing a private (non campus backbone) path for NIS and DNS services to the various other machines (on their NFS Ethernet ports). There is a third Sparc2 that is used to support the Xerox Soleil product that allows the Xerox 4090 printer to print either PostScript or ASCII data from a TCP/IP network.

This collection of servers, and dedicated Ethernet ports along with the Auspex file server spread the I/O across enough paths to allow things to work.

- Magnetic tape support currently resides on a VAX running VMS. This is because we were not able to find any UNIX software that could read all the various kinds of IBM standard labeled tapes. Data coming in on tape is read in to the VAX and then either transferred to 8mm tape on the VAX or ftped to the Auspex file server. At present the users that used to run their large data sets directly from tape, either store subsets on the file server disk and copy off tape as needed (a time consuming job) or do without. We will be installing a HP 20 gigabyte optical juke box onto the network in an attempt to provide users an automated (if low performance) way of storing their large data sets.
- An unanticipated advantage of having the Auspex file server turns out to be system backup. We have dedicated two Auspex disk drives to system backup. When a full backup is to be done, the two spare drives are mirrored to the active file system. This process takes about 20 minutes to make a duplicate copy of the online file system. When the mirror is complete, the mirror volume (now identical to the online volume) is detached from the mirror, which gives us a current snapshot of the online file system without having to bring the system down to single user mode. At this point the detached mirror disk is fsck'ed, and then backed up (presently using dump, but we may switch to doing a dd of the volume) to a labeled 8mm tape using some shell scripts we wrote to verify that the tape in the tape drive is the tape it should be, by reading the label before doing the dump. Daily incrementals are done on the running file system, so it is possible that we may lose up to a week's worth of data in the worst case. A shell script tape directory program enforces a dump cycle of 14 daily incremental tapes, and 4 level 0 weekly tapes, and 4 level 0 monthly tapes allowing us to recover data from up to 4 months in the past. This directory presents the required tape label to the backup script to make sure the correct tapes are being used for a backup.
- On the Sun Sparc server machines (NIS Server and DNS server), SunSoft's Backup Co-Pilot product is used to give us consistent dumps of the NIS and DNS machines without having to shutdown.
- Tape library services have more or less fallen back on a manual system. One VMS product has been tried, but was found to be unsatisfactory. We are being saved at the moment because tape use has fallen off significantly because of the difficulty of using the present tape operation.
- BitNet. Since we have some users that still need BitNet, we chose to implement BitNet on a VAX running VMS, using the Jnet BitNet software and a package called MX from RPI to implement a BitNet to TCP/IP gateway. Bitnet file transfers have to be done to and from an account on the VAX.
- The public X.25 interface is also done on the VAX, using a DEC X.25 Router2000 gateway server and a software package called PSI which provides access control and accounting (both of which we need in this application!).
- Printing. The central Xerox 4090 printing service has been replaced by 6 HP3si laser printers, driven from a Novell server that accepts jobs from all hosts (our UNIX hosts, plus Macs, PCs and other people's UNIX hosts on the campus backbone). The advantage to the central Novell server is that jobs can be spooled from anywhere to the server, and then be released via a release terminal (an IBM pc) that is next to each of the 6 distributed servers. This means that you can print your job from the closest of the printers. In addition, there is a magnetic card reader attached to each of the printers that has 5 cents decremented for each page printed on the printer meaning that the printing is accounted for as well. These cards can be charged up at several vending machines around campus allowing you to print if you run out of money after business hours.

  As mentioned earlier there is also a SparcStation running Xerox's Soleil software, that allows large and/or multicopy PostScript or ASCII jobs to be printed on the Xerox 4090 printer in theory at the rated engine speed of 92 pages per minute (in practice there are still a few bugs and we don't get anywhere near that throughput).

An interesting sidelight, the Xerox 4090 in MTS days used to print in the range of 600,000 to 800,000 pages a month (combined MTS and administrative VMS printing) at the present time it is doing about 200,000 pages a month, a large portion of that from the VAX. This drop off has not all been due to UNIX and the HP3si printers (the administration systems are also printing less); we believe that many of the course handouts that used to be done on the 4090 under MTS are either not being done (i.e. the file is made available online for the students to print themselves) or possibly are being done by the print shop. At least some of that demand has done other things for the summer but will return to the 4090 for the fall semester. (This may cause some interesting times for us in the fall.)

- Accounting. luckily the accounting for anything other than hard dollar costs (in practice public X.25 connections and printing) was ruled no longer required nor useful by the task force, so we are fine.
- Security. Mainframe operating systems in general and MTS in particular tend to have less security problems than UNIX. Some of the reasons are: the operating systems are proprietary and there is not general access to the operating system source code. Security tends to be far more important to the mainframe customers, and therefore more of a marketing issue than has been typical in the UNIX market place. A security breach on a mainframe system providing service to a large corporation can have a severe monetary impact, either through fraud or the loss of trade secrets.
- User directory. This was replaced by an X.500 directory system based on the QUIPO software, with locally written code to allow a lookup from Macs (and in the future PCs?). This runs on one of the Sparc2 servers (the DNS machine), with 64 megs of memory, and contains some 25,000 names at present.
- E-mail. The E-mail system was replaced with several options. Part of the problem here is that in the MTS case, since mail was on the central machine, mail could be accessed from any of the communication channels (directly connected terminal, dialup line, public X.25 connection etc.). The challenge with UNIX was to maintain this diversity of possible connections, while allowing mail access to move out to individual machines if desired. This was achieved by having a central mail host that accepts all incoming mail to an address of the form user_name@sfu.ca, and then directing that to the users central UNIX ID. If the user chooses to get mail on his or her personal machine, then they set up a .forward file to that machine on their central UNIX account directing mail to their personal machine. For people that choose to read mail on the central UNIX hosts, we support the Elm E-mail package on all of the central hosts. For people who choose to read their mail from a Mac we support the POP protocol with UNIX pop server on the mail host and for the Eudora Mac POP client. We have made some modifications to Eudora for use over dialup and public X.25 connections that have been fed back to the author. PCs are somewhat less supported; there is work going on to improve PCpop (and now NewPop) to get POP support on the PC. These changes too are being fed back to the authors. For those PC's connected via Novell we run the Clarkson Charon gateway on a Novell server to provide mail access. There is also work going on on providing interfaces from the various platforms to the X.500 directory service to provide E-mail address lookup from within the various mail packages.
- Conferencing. The Department Of Education commissioned a professor in the Communications Department whose speciality is electronic conferencing to examine the available conferencing systems and recommend one. Parti from Participate, was the package chosen, and it was installed on the general login server for teaching and research. Some of the former Forum users considered Parti to be a step backwards in conferencing and instead chose to do their conferencing using NetNews (and complained loudly and long about the money spent on Parti, but not to the committees that made the decision!).
- The PDP11 based terminal interface was replaced by three Annex3 terminal servers from Xylogics. We chose these terminal servers because the interface was UNIX like (as opposed to VMS like as some of the others were), they are capable of requiring a password for access, and the source to the UNIX side authentication daemon is supplied so we could (and did) modify the authentication to access the NIS server to verify passwords. We considered the Xyplex terminal servers which do authentication using the Kerberos protocol, but since Kerberos had been ruled out as too complex to be done within the deadlines imposed, this option was not chosen.
- Education. There was of course a massive education problem, since neither the systems people nor the user community were experienced in UNIX. This was solved by having the people that produced the MTS documentation create UNIX documentation. This takes the form of a series (more than 30 at present)

of single page handouts called "HowTo"s. Each HowTo covers a specific topic or UNIX command in detail or covers how something that the user used to do on MTS is now done under UNIX. The PostScript code for the HowTos are stored online along with a program that will send a copy to one of the printers, and preprinted copies are kept in racks in the Computing Center to be picked up. At present we provide ascii and PostScript versions of the HowTos that can be obtained by anonymous FTP and via the campus wide information service (currently provided by gopher).

- User account creation. This was done initially via shell scripts, using the NIS maps as a pseudo database for deciding whether or not a UNIX id or a mail alias was already in use for another account. All accounts were created, along with their home directory and initial files on the file server and a randomly generated (or supposedly randomly generated!) password set. This took a lot of time to process, but the scripts could be (and were) created in a couple of days, and were modified on the fly as things broke. Minimal function (due to time limitations) was put in, accounts could be created, but deletion and changes were not, these functions were to be added later.

What We Did Right

In hindsight, these things went well:

- Wired the campus. We were lucky enough to have technical people with a vision of the future, and management that felt that computing was an important part of the university to approve and fund the installation of the campus networking infrastructure. What is more, the design that was made when the choices of technology were no where near as clear cut as they are today have withstood the test of time, the infrastructure that we have in place is (as it was designed to) capable of supporting the new technologies as they emerge today.
- Committed to Cabletron equipment as the network vendor of choice for the network hardware. As with the cable plant, the Cabletron hubs have grown to support in a modular fashion the new technologies as they emerge, protecting our investment in network hardware. Their equipment has been very reliable, and the support has been good.
- Issued a "visual RFP". Since we had little idea how to capacity plan a UNIX system, we instead provided the vendors with our requirements as number of logged on users and what we expected them to be doing, and required the vendors to quote a system that met those requirements. Since our site was seen (and in hindsight, correctly so!) as an important example (at least in Canada) of how the conversion to UNIX would go, the vendors had a lot of incentive to make sure what they quoted would do the job.
- Purchased an Auspex NS5000 NFS file server. As pointed out earlier, the Auspex people appeared to be the only vendor that we talked to that both understood the I/O issue, and had a working installed system. Several points worked in Auspex's favor, as we mentioned: their marketing presentations were first rate, technical questions could either be answered by the presenter, or forwarded to the technical staff and an answer was forthcoming. The most compelling reason, however, was not the Auspex marketing folks (who could be expected to praise the product to the skies!) but in fact the customer reference sites. Many if not most of them, had more than one Auspex, strongly suggesting that they were happy with the machine. At one site we contacted, the Auspex maintainer had to think for a while (when we asked if the machine crashed a lot), then said some thing to the effect of "well, the only two I can think of were both power failures, and we were one of the beta sites and have had one for a year or so". To say the least, we were impressed by the obvious reliability of the box if it required thought to figure out when the last crash had been. At this point we have been running our NS5000 for a little more than a year, and by and large our experience has been the same, there have been problems that caused crashes, but not a large number of them.

  At the point that we bought the NS5000, we had not considered backup particularly (and therefore the mirrored backup didn't enter into the purchase decision). At this point, if we had the decision to make again, the mirrored backup capability that the Auspex can provide (especially given the conclusions in Elizabeth Zwicky's paper on the backup torture test) would be one more point in favor of buying an Auspex again.
- Selecting Silicon Graphics machines as the research machines and the general login server has worked well for us. The local SGI office provides us with excellent support, and the machines have performed as advertised. We would note that the general login server that was specified to support 80 users has seen a peak load of 110 users, and at that point response was still very good. This of course depends on a rigorous enforcement of no CPU bound jobs running on the login server, even a

single CPU bound job will tend to slow response on the system down, and several make it close to unusable.

- Set a UNIX novice to doing the documentation. This turned out to be a very good move, the documentation that was created covers the points that the novice user needs to know (since it was being written as the person writing it learned UNIX!), and yet is full and complete (our UNIX consultant learned of a command option he wasn't aware of while reading one of the HowTos). The HowTo collection is available via anonymous ftp (see the last paragraph), all we ask is to be given credit if you use them.
- Used VMS for several areas that UNIX supports poorly or not at all. We are lucky enough to have a VAX for the administrative computing, and in fact had a spare one that was surplus to the administrative system's needs but not worth much on the open market. This machine provides us with BitNet support (using Jnet and MX software packages as noted earlier), controlled, accountable X.25 public dial access using DEC PSI software. This has become important, because our public X.25 bill has tripled over what is was on MTS, largely due to the fact that MTS was line mode oriented, and a single packet contained a full line of characters, and UNIX is character oriented and likes remote echo (and therefore often costs two data packets per character rather than one packet per line), and we get charged by the kilopacket. The accounting software is allowing us to evaluate who is using the service and charge the cost back to the departments involved.

  In addition, this machine supports a bibliographic package called BRS that is being used to replace a package called Spires that ran on MTS but is not available on UNIX (we understand there is now a UNIX version of BRS). Perhaps the major use of VMS is to support the conversion of tapes. We have around 10,000 IBM labeled and unlabeled tapes from the 15 years of MTS in a variety of different formats (some of them custom). As well various of our researchers and our research data library get statistical and other data from other sites and governments on "IBM" style tapes and wish to process the data on UNIX. We were not able to find a UNIX based package that could read the various formats of IBM labeled tapes, but we were able to find several programs that run under VMS that can (with greater or lesser difficulty) read these tapes. As well, our VAX already has both 3420 (or "round") and a pair of SCSI 3480 (or "square") tapes, and the SCSI bus meant that adding an 8mm and cartridge tapes that UNIX supports could be done.

  The bottom line here is don't get hung up on doing everything on UNIX, look around and see if there is a more appropriate platform for doing some of the things UNIX can't.
- Using DEC's VAX Cluster Console package to provide the operator interface to all the UNIX hosts. This package again runs under VMS in this case on a VAXStation 3100 color workstation, and uses a Xyplex LAT terminal server in one of the Cabletron hubs to make dedicated connections to the terminal server ports that connect to most of the UNIX systems' console ports. This allows a single screen in the operator area to monitor and control all of the systems, it logs to disk all console messages for all machines, and allows the systems administrators to access the console of any UNIX box remotely (i.e. over a dial up line from home). In addition, there is a process that scans the console data as it comes in and that can be programmed to recognize error messages and produce an alarm condition on the operator's console (typically set the icon for the affected machine red, but it could page a system administrator if desired).

Some things, though, did not go so well:

- Magnetic tape support would have to head this list. There are two issues on tape support: transfer of data from off site sources for use on UNIX, and affordable, fast support for the processing of data sets that are too large to fit on disk. Of these two, the VAX solution detailed above works for the data interchange case, but at a large cost in skilled manpower to identify the tape type and arrange for the correct program to read and transfer it. In the MTS case, most of the time MTS could read the labels and figure out for itself what the tape format was and provide a data stream to the user without computing center intervention. The processing of large data sets directly from tape (bypassing disk) has not yet been solved. The performance of tape units on UNIX is a small (and unusable) fraction of the performance of a mainframe tape I/O system. Even if the performance were tolerable, there is no support in UNIX for tape management (in terms of controlling access to possibly sensitive data on the tape by user id), or support for the operator interface to request that the operator mount a tape on a tape drive and dedicate it to a user (we are aware of the packages around that will do this). We are at present installing a 20 gigabyte HP optical juke box system to attempt to solve the large data set side of this problem, and several sites

in the MTS community are considering a joint effort to write a UNIX version of IBM tape support (since we had to do so for MTS, we know how).

- We underestimated the requirement for disk on UNIX. There was about 10 gigabytes of user data (out of a total of some 16 gigabytes in total) on MTS, so we bought a 10 gigabyte Auspex file server. We are presently at 18 gigabytes on the file server and will probably be adding more space. It looks to us like doubling your mainframe disk capacity when going to UNIX is just about right for planning purposes. One of the other MTS sites that had made some movement towards UNIX warned us that this is what they had found, and it appears they are correct. It is not clear to us why this is so, and it may only apply to sites running MTS (since both sites that have seen the requirement run MTS). Certainly some of the increase (perhaps all of it) has been caused by data that was stored and operated on from tape moving to disk in the UNIX case. There was no time to identify the cause, since the solution was simple: buy more disk.
- We didn't have time to make a proper user account creation system. This showed up as part of the next point, security (or more correctly the lack of it). Given the time frame imposed, this was unavoidable. There was simply not time to create such a system and still meet the deadlines, and the shell script and NIS database method worked – the accounts got created. The problems are several: the shell scripts never made it into sufficient control to allow the deletion of unused accounts, so many inactive accounts are active on the system. The random password generator wasn't quite random enough. It generated some 15 or so identical passwords for different accounts. In hind sight (which is of course always 20/20!), we should have checked that the just generated password did not match any previously assigned one. The correct (or at more correct) answer to this is to have a database (Sybase, in our case) hold all of the data about an account, including the id and mail alias to allow checking for duplicates when a new account is created, and only generate an account and assign a password when the user requests use of the account and identifies him or herself. This avoids the problem that we currently face of having many accounts that were never used until they were broken into and stolen.
- We failed to address the lack of UNIX security. This was probably our major and most expensive error. As we have pointed out before, MTS is a reasonably secure system, in addition, since there are only 8 MTS sites (7 now that we are gone!), there weren't a lot of people that knew what to do should they break a password and get in. Neither of these issues are true on UNIX, and especially not if you are connected to the Internet. Our lone UNIX expert is very experienced at securing UNIX systems, and therefore our hosts are reasonably secured, and we have been tightening up host security. We were aware that NIS and NFS were not very secure, but given the deadlines, we had to accept that they were the only way this was going to get implemented. AFS would have been a more secure and possibly better approach to file services, but it is not supported on the SGI machines, and as we pointed out earlier, there is NFS expertise on campus but no AFS expertise, so NFS was the safe choice. We have installed a copy of npasswd to force the selection of better passwords (after an initial run of crack broke some 800 passwords), and hope to install shadow passwords or Kerberos on all machines, at which point we will revalidate all users of the system and not automatically create an account for each user, but rather have an entry in the database that can instantly create an account, home directory, mail alias etc. when the user requests one and presents id.

  Chasing crackers and recovering from attacks on the system have chewed up a lot of staff time that no one expected to have to spend. There is an additional problem at SFU: Since MTS was a secure system, many of the users were used to storing confidential data on it and assume the same is true of UNIX. This is, of course, not so, it seems every day some new way to break security on a UNIX host is found and published.
- We did not (or could not) budget for more staff to support UNIX. The same number (45) of people that supported the central MTS system are now attempting to support 16 UNIX hosts from 3 (soon to be 4) different vendors, as well as provide UNIX support to the campus. This is clearly not possible, and what has fallen by the wayside is user support, since the manpower that used to be devoted to doing this is now going to keeping the UNIX systems running and maintained, and trying to document it all. There was a request from one of the lower level committees for an additional 50 staff members that would be distributed out to the the departments to provide UNIX support at the departmental level (there are some 60 people in the central site at present). Clearly, we needed to budget for more manpower to support a distributed computing effort (as opposed to a central

mainframe). I would note that we have not distributed very far, all the server machines that we support are in fact physically in the same machine room where that mainframe used to be. If we had in fact moved them out to the various departments, then there would have been an even stronger requirement for more manpower.

UNIX "Features" Attempted To Drown Us

We discovered a number of things in UNIX that work fine when you are dealing with a single workstation with from tens to hundreds of users, but that cause large problems when you are dealing with 18,000 to 25,000 users.

- If sendmail can not resolve the address to a user id, it proceeds to walk the NIS password map looking at the GCOS field trying to find a match so it can deliver the mail. When there are 18,000 accounts, several things happen: often, more than one message arrives at once and there are several processes attempting to walk the map. Since there are 18,000 entries in the map, a single walk takes a long time. Even one map walk provides a very heavy load on both the NIS server and the Ethernet over which it is communicating, overloading both the network and the NIS server and bringing the whole system to a grinding halt (as no other machine can get NIS service). The solution to this is fairly easy: throw away the vendor supplied sendmail, and install either IDA sendmail or Berkeley sendmail with the map walking code commented out. We chose to use Berkeley because we already had a sendmail.cf, and we had some troubles getting IDA to work (and the systems were dead NOW!).
- NeXT machines like to look over the full password file every 15 minutes or so, which again when it is 18,000 to 25,000 names long causes a heavy network load. When you have a whole lab of them boot at once: disaster. Again an easy fix, disable caching by setting the interval to 0. Finding this before your network melts down is much nicer than finding this is the reason that the whole system has been dead for the last hour or so.
- Our current password file is approaching the 26,000 mark (partly because the shell scripts that add accounts have no corresponding removal of dead accounts yet!), and at this level, you can cause the ndbm program to attempt to create a sparse file that is greater than 2 gigabytes (which of course fails). The solution we used here was to reduce the amount of text in each password entry (primarily changing "*disabled" to "*" in disabled password entries), but it indicates a limit that sites with large numbers of users need to be aware of.
- Due to our "one account accesses all" policy, we have exceeded the 32767 mark with some of our UNIX uids (since we have a common uid space for all machines on campus that use our file server). Several of the UNIX versions we have can not deal with a uid greater than 32767, and this is another limitation that sites with large numbers of users need to be aware of. We believe that we have most of our problems with this worked around. We would note that the maximum uid is 65535, so this may be a problem for very large sites.

Work in Progress

We are still working on some things:

- Modify the RPI database driven account management and generation system to work at our site. This is in progress now and should be complete by the time this paper is presented.
- Merge in a database driven resource accounting program that is being used to charge for printing on the Xerox 4090 and the departmental laser printers in the Computing Science department. RPI is interested in implementing this at their site.
- Install a 20 gigabyte HP optical jukebox and management software for the storage of very large research data sets. This is on order, and should be completed by the time this paper is presented.
- Install a public lab of 40 NeXT stations, and resolve the security and support issues surrounding this given no additional manpower. This should also be complete before this paper is presented.
- Implement the OnLine Consulting (OLC) from Project Athena at MIT. This is up now in test, without any of the standard Athena services (Zephyr, Hesiod, or Kerberos). The Discuss conferencing system used to store completed conversation logs has been replaced by an interface that posts the done logs to NetNews, and these changes will be sent out on the OLC developers' list.
- Convert our network from a bridged network to a routed network. For the last several years the networks of the 3 BC universities have been connected together as a single, large bridged Ethernet (with the intercampus connections being via T1 links). Apple tells us that we are the largest Appletalk / Ethertalk network that they are aware of. All three universities are now moving toward a fully routed topology. We are partly being driven to this by security concerns and problems: A Mac set to the same IP address as the CA*net

router blocked Internet access to all three universities until it was found, and of course, a lab of 40 NeXT machines is going to need some heavy securing (with being routed to the backbone being the start of that!).

- Install a donation of 20 X terminals from DEC. These X terminals use a virtual memory scheme to a DEC controller both to keep network traffic down and to allow less memory to be used in each X terminal. This will be a pilot project that may or may not lead to more X terminals being deployed.

Future Work

This is a list of the projects that we would like to undertake in the future to make computing life on our campus better; budget and time constraints will, of course, modify this list.

- Install Kerberos for network security. This is expected to be a large and hard to maintain job (unless of course the vendors start to support Kerberos).
- Install the shadow password suite on all of our systems. This is made more difficult by the use of NIS, so we may reconsider the use of NIS (although recent changes from Sun have made the master more secure.)
- Install secure 10BaseT hubs in the various computing labs to prevent the lab machines (Macs, PCs, NeXTs etc) from sniffing data from the Ethernet. A secure hub is one that only sends the data to the source port and the destination port; all the other ports get the packet, but the data portion is replaced by either random garbage or blanks so the data is not visible.
- Join in the effort to write an IBM tape support package for UNIX with several of the other MTS sites.

Conclusions

We have learned many things:

- There can be computing life after the mainframe: a diminished, I/O poor life perhaps, but life nonetheless. A large majority of the users on our campus use computing only for E-mail, and for them life has gotten better in many cases. At least some of the people out there use Eudora on the Mac, and never (and don't want to) sign on to UNIX. We discover this when Crack finds their password and they don't know how to sign on to change it. Replacements for the standard statistical packages (often a UNIX version of the same package) have kept the statistical users happy, with the exception of those who used large datasets from tape in the past. In general researchers running programs in Fortran were able to convert over to UNIX (not without effort and grumbling, but they were able to do it). There were a handful of users that had applications that are very MTS specific, and in fact a handful of them are still running on MTS at one of the other sites as being more efficient than converting. This is a point to keep in mind if you are converting, line up another site that is still running whatever you are converting from to satisfy those few users who really do need whatever you are converting from but only for a little while longer (making the cost of conversion too high!).
- If the full cost of the conversion is taken into account, we don't believe that a distributed UNIX system is cheaper than a mainframe; in fact due to the manpower increase, we think it is more expensive. (The full cost of the conversion includes the support cost of those people who chose to "do it for themselves" and run their own UNIX workstations, thus shifting the cost from the computing center budget into a departmental budget – possibly by funding a "research assistant" who just happens to be a full time UNIX system administrator).
- The Director Of Academic Computing Services here keeps saying that we should be giving away CPU cycles since they are cheap, and getting cheaper, and charging for the people required to maintain the systems who are expensive and getting more expensive, an exact reversal of the original reason for mainframes (when the hardware was expensive and the people were, relatively speaking, cheap).
- It is probably true that the increase in what the users can do with the new systems in terms of being able to run or share code from and with other sites on the network, graphics, the ease of use of e-mail more than justify the cost incurred by the conversion.
- Your VAX system programmer is your friend. Many of those tasks that are hard to do on UNIX will fit nicely on his or her VAX under VMS. Stress much this will increase his or her job security, just make sure that you forget to mention how much extra work this is going to be.

  The real conclusion here is don't assume that a task can only be done by the UNIX system, look at the other resources that you have, and implement a service on the most appropriate platform. In our case, many of the instructional tasks that were once done on the mainframe are now done on either Macs or PCs in either public labs run by the computing center, or private labs owned and supported by a department that has sufficient demand to warrant the support expense. Be prepared to look at cost sharing plans where the

department buys the machines and provides the space and the computing center provides a part of a full time person that is shared with other departmental labs because there isn't enough work to justify a full time position.

- If you make use of IBM labeled tapes you are almost certainly going to have problems moving to a UNIX system. Support for labeled tapes and mainframe like tape access control does not seem to exist on UNIX.
- If your site is connected to the Internet, you are almost certainly going to have intruder problems unless you take the time to guard against it before you install. Installing a password checking program like npasswd or passwd+ to force users to set good passwords is a good start. Get and apply the latest Sun patches that allow you to restrict access to your NIS server to hosts you select. Consider installing the shadow password suite if possible, consider installing a firewall machine between you and the Internet. Be aware that as shipped most UNIX boxes are wide open security wise; hire an experienced UNIX system administrator and listen to his advise when he suggests shutting off most of the "r" commands and sending mail to a program via sendmail. Even if the system would be easier to use with these programs enabled, they are not secure. Initially we did only the last of these, and thought our system was reasonably secure. We learned differently several times, and have lost a lot of staff time and are still losing a lot of staff time tracking security problems and attempting to close holes.

Availability

This section describes how to acquire many of the things described in this paper.

The HowTo documents (in PostScript and ASCII for Gopher) are available for anonymous ftp from ftpserver.sfu.ca in /pub/docs. All we ask is that you give SFU credit if you use them. The contact for HowTo questions (or contributions!) is Margaret Sharon (margaret@sfu.ca). The originals are in FrameMaker on the Mac.

The code that does mirrored backup to labeled tapes on the Auspex file server is available (if a bit rough!), contact Peter Van Epp (vanepp@sfu.ca) if interested.

The lpr filters and accounting software for the Xerox Soleil product can be made available if anyone else is using Soleil, again e-mail Peter Van Epp (vanepp@sfu.ca).

The database driven account management system is also probably available with consultation from RPI, please send e-mail to Richard Chycoski (richard@sfu.ca).

The Novell solutions, both for distributed printing and for reserving machine time in assignment labs can probably be made available send e-mail to Lionel Tolan (lionel@sfu.ca)

Advice, (possibly worth what you pay for it!), if you are facing a conversion like this we can answer e-mailed questions and arrange a limited number of site visits.

References

Baines, Urquhart (editors), *You Really Can Get There From Here: SFU's Migration to UNIX from MTS*, Proceedings Of The Community Workshop 92 (paper available via ftp from ftpserver.sfu.ca in /pub/ucspapers/RPIPaper92.ps.Z).

Zwicky, *Torture-testing Backup And Archive Programs: Things You Ought To Know But Probably Would Rather Not*, Proceedings USENIX LISA V, pp. 181-185.

Kolstad, *A Next Step In Backup And Restore Technology*, Proceedings USENIX LISA V, pp. 73-79.

Shumway, *Issues in On-line Backup*, Proceedings USENIX LISA V, pp. 81-87.

Polk, Kolstad, *Engineering A Commercial Backup Program*, Proceedings USENIX LISA V, pp. 97-103.

Author Information

Peter Van Epp has been a staff member of Computing Services Operations at Simon Fraser University for 5 years. Prior to joining SFU he spent 6 years at an airline as a systems programmer on the mainframe based reservations system. The previous 10 years were spent in a variety of real time control companies using mini and micro computers. He can be reached electronically at (vanepp@sfu.ca or vanepp@SFUVAX.Bitnet).

Bill Baines has been a staff member of Computing Services Operations at Simon Fraser University for 5 years. Prior to joining SFU he spent several years supporting administrative and real-time computing at a large industrial site where he learned more about performance management and capacity planning issues than he ever wanted to know. He has a B.Sc. degree from the University of Waterloo and can be reached electronically at (Bill@sfu.ca, or bill@SFUVAX.Bitnet).

Is Centralized System Administration the Answer?

Peg Schafer – BBN

ABSTRACT

The old standard model of centralized system administration does not fulfill the requirements for many large sites. What are the alternatives? Presented is a discussion of the problem and a proposed model of distributed system administration. I believe the future of system administration for large sites does not lie solely in the development of centralized services. Rather, it lies in the co-operation of central services with local system administrators who, in turn, provide the primary support for their user groups. I propose a model by which administration responsibilities are shared between a central group and a local system administrator for each group.

I believe there is a role for centralized administration; at the same time I firmly believe there are a range of services which can only be supplied efficiently by a local system administrator. This paper will suggest roles for "Central Services" and "Local System Administration". The Central Services group provides support for the services which are of common use across the total environment and provides information and support services to the Local System Administrator. The Local System Administrator is responsible for the efficient adaptation of the machines to the computational task of the group. The most important contribution this paper has to offer is the notion of "Distributed Responsibility" of system administration.

Today's Standard Model of Centralized System Administration

Today's standard model of centralized system administration originated in the ancient days of "THE" big mainframe in "THE" machine room. The theory and practice of centralized control has been the foundation of policy development, user services, hardware configurations, network management and software development. Indeed, the concept of centralized administration is so basic, it is the default mode of thought in discussions of system administration.

In Favor of Centralized Control

Historically, there have been many advantages to centralized control. As cited above, the initial working model revolved around a single mainframe. All knowledgeable people were housed together, facilitating one method (style) of work which could be communicated among the whole group and to new workers. Users willingly utilized this one repository of expertise, as it provided one point of contact for requests, complaints and questions. As sites grew, and machines became more numerous, new policies sprang forth based on the single main frame policies.

As support services enlarged to meet the needs of the new distributed technologies, specific areas of expertise developed. A centralized support group will now consist of experts in separate areas of system administration, e.g., network, news, mail, nameservers, Appletalk, etc. Indeed, many centralized support groups have "internal development groups" which do not support users, but create the necessary toolset required for centralized support. But let us not forget the most important reason for centralized support: *It is cost effective.*

The Dark Side of Centralized Control

Despite the advantages stated above, centralized control may not be possible or the best solution. Often, a site may have had at one time centralized control, but now, in reality, is a weak dictatorship prone to confused disorganization. Without benefit of organization or any real authority, users and groups will do what they wish, resulting in disorder. On the other hand, there may be a strong centralized support system which maintains one standard system configuration for a large portion of the total machines. Deviations from this standard system configuration will present great difficulties. Non-supported systems are left free floating – lost at sea. Much of the current literature on system administration deals with the problem of standardizing systems.

Another difficulty is that the model of central control may not match up with the corporate structure of a company. Due to internal conflicts over goals, resources and personnel, some divisions may resemble "The Warring States of Greece" rather than one company united in a common pursuit. Do not underestimate warring managers as a road block to efficient system administration.

Finally, a rule of thumb in programming is: The larger a system is, the harder it is to understand, modify, debug, fix and speed up. The same rule can be applied to a large support group. Simple little questions or a 30 second fix may require a standardized 24 hour turnaround period. Generalized questions become hard to answer. Specific questions on local applications may be routed through a number of people before the answer is achieved. Users become unhappy with the impersonal contact (i.e., phone or e-mail, not face-to-face). Users may interact with a different support provider, developing continuity difficulties. And let us not forget the difficulty of physical space separations. Physically walking over to building 34 from building 3 to reboot a system may waste 30 minutes to an hour.

Diversity is Now the Name of the Game

Today, centralized support groups are required to maintain many types of systems in every nook and cranny of a company – systems which represent the many diverse uses of computers. Centralized control may not be able to meet the specific needs of these systems. These needs may not have been considered or may not conform to centralized policies. Cookie-cutter techniques and cloning methods may not be adaptable to the variety of hardware platforms and unique configurations. Let's look at some examples.

A company's research area will have specific requirements. Their machines are the newest of the new with a high turnover rate. Depending on the area of research, machines will have special purpose (and often VERY buggy) hardware and software. Realtime systems, graphics and medical research are good examples. This research user group consists of experts who may know more about computing and hardware than do the support personnel.

Software developers are driven by release schedules. They are very sensitive to the timing of any system changes. Their machines range from beta software and hardware platforms to the oldest of the old. They will have at least one of everything; a very heterogeneous environment. Again, the user community here may be very knowledgeable in some areas.

MIS departments are completely different animals altogether. Their management structure may view computers as "black boxes" which they utilize to get their very important work accomplished. They may have incompatible applications and architectures, and an insistence on holding on to old standard methods of work. Change or innovation requires extensive examination and thought. Accounting schedules are tight. Security is always a concern on these machines; not only from outside attacks, but strong measures must be followed to prevent intrusion from within the institution. In addition, unlike the user communities discussed above, this user community may consist of a large population hired to sit in front of a terminal to interact with a specific application, i.e., basic users.

This diversity I have been discussing is not limited to differences in group functions and user communities. In addition to their use on desktops and as workstations, many companies use computers in their manufacturing divisions, and system administrators are increasingly called upon to support these machines as well.

Accounting and security practices also can vary widely. If your company has government contracts, their methods of administration and security concerns are guaranteed to diverge from non-government areas of the company. Divergent security requirements among groups may also conflict. All this serves to heighten the complexity of centralized administration.

Some departments may utilize a "total package" for their work. A documentations or publications department which utilizes Framemaker or Interleaf is a good example. A VLSI group will utilize ornate CAD/CAM systems. The managerial and secretarial staff may tend to use MAC-based business applications. Turnkey systems are everywhere, and need to be supported. It's difficult for a centralized systems administration group to support a very broad range of applications software.

Ownership of machines within an organization can contribute to the diversity faced by centralized system administrators. The support task is greatly simplified by a homogeneous collection of hardware platforms. However, different groups or divisions in an organization are most likely to purchase equipment tailored to their own needs, so within such organizations, there is little hope of finding a standard hardware configuration.

In addition to this diversity, corporate structures can be barriers to centralized control and support. Management's conscious decision not to allow "outside" control is a valid alternative; control and guaranteed levels of service will be internal to the division, group, etc. Never underestimate the thirst for freedom of action.

Funding is always a determinant. Different groups may have their own sources of income. One group will have money to buy new machines, while others may have to cut back. Again, this may make it extremely difficult to standardize system configurations across the organization.

In sum, as centralized system administrators are required to support an ever increasing variety of machines, applications and groups, they are finding it extremely difficult to fulfill the operating requirements for all diverse groups.

A Proposed Model

I believe the future of system administration does not lie solely in the development of central services. Rather it lies in the co-operation of central services with local system administrators who, in turn, provide the primary support for their user groups. I propose a system by which administration responsibilities are shared between a central group and a local system administrator for each group. I believe there is a role for centralized administration. I also firmly believe there are a range of services which can only be supplied efficiently by a local system administrator.

Let us start with some basic definitions:

- **Computing Community**: An interacting population of individuals of all skill levels, or a group linked by a common policy. In our case – a large company.
- **Computing Environment**: Very much like a user's computer environment, but on a larger scale. The machines and their configuration.
- **Central Services**: A group which provides services which are of benefit to the total corporate computing environment and in particular the needs of the local system administrator.
- **Local System Administrator**: A person or persons who work within a group, and are completely familiar with the computational requirements of the group.

The Proposed Role of Central Services

Central Services (CS) provides services which are utilized by the total computing environment. Central Services organizes the diverse groups and supplies them with information necessary for their continued development. The CS must direct its efforts toward the the development and enhancement of the computing environment. Presented are a few of the working areas for CS.

Policy and standards generation is a key objective. CS is responsible for development of the policies in conjunction with the local system administrators and the computing community. Timely revision, enforcement, and arbitration responsibilities fall to the CS. Policies and standards can cover every facet of system administration from policies on security to standardized rc files. Standards in areas such as "file system layout" and naming conventions can be agreed upon and distributed by the CS. Local system administrators may deviate from these standards as required to fulfill local computing requirements.

The network is the life line of computer environments. Network administration, both hardware and software, belongs in CS. This group would maintain the gateways to the world, and critical network boxes. All global network services fit in here – Domain Name Server, NTP Servers, etc. Large databases accessible by the whole company are maintained by the CS: NIS global administration, company phone book, information services, dictionary servers, finger databases, libraries, license servers, etc.

E-mail is a service to the total computing community, hence anything connected with it is under the domain of the CS: mail machines, sendmail.cf, /usr/lib/aliases, etc.

Distributed printing services are a big headache everyone is willing to give to anyone who will do it. As one manager put it, if you were evil in a past life, in this incarnation "they put you in charge of the printers." In this model the CS would be responsible for the configuration and maintenance of the printer hardware and software as it is another service to the whole computing environment.

Security is of great concern for all. The CS group can monitor the network, spot check machines, advise the community on new security features, coordinate security alerts, evaluate new operating systems for security configurations, advise local system administrators on the proper security procedures, provide a site representative to CERT, etc.

Information gathering and dispersal to the user community is critical. CS may publish a computing guide for users. This guide would contain pointers for more information, advise on the resources available, explain policies, etc. But the CS should also provide forums for discussion of critical issues by the user community.

The organization of vendor presentations and evaluations is very helpful. Not only can the CS examine and evaluate new hardware and operating systems, they may host these evaluations and advocate evaluation by members of the computing community. Indeed, it is likely there are members of the user community with relevant expertise; their evaluations and recommendations can be solicited and distributed.

Since computing technologies are turning over at an ever increasing rate, continuing education of the computing community is essential. The CS can coordinate classes, tutorials and presentations on many topics including: new methods of computing, new system administration techniques, etc.

Members of the CS may serve as official representatives of the company's interests in external organizations having to do with industry standards and user groups, (e.g., POSIX).

Backup and tape archival services are classic CS services.

Site licenses and contracts for software and hardware can be handled efficiently by the CS. Negotiating for a discount based on volume is always welcome.

Some groups request help in the evaluation of their computational requirements and selection of hardware for purchase. This complements nicely the vendor presentations and evaluations cited above.

CS Support for the Local System Administrator

In addition to the functions listed above, CS communication with and support of the local system administrator is vitally important. The CS can develop software tools for the local system administrator such as automated installs, site specific accounting programs, new user scripts, etc. The CS can provide backup support for local system administrators. A "visiting" system administrator can be on hand while the local system administrator is out for conferences, training, vacation, etc. The local system administrator can rely on CS for help with a particularly perplexing problem; sometimes two heads are better than one. When 25 new machines just roll in the door "high tide" services are always welcome.

The Role Of The Local System Administrator

What is left for the Local System Administrator (LSA) to do? Lots! An emerging role of system administration is now in the management of local application software. Often the local system administrator is the "tool maintainer". As mentioned above, machines and the applications software have diverged to such an extent that detailed knowledge of the utilization of the machines is a key factor in day-to-day maintenance and problem resolution. The LSA must have a detailed understanding of the layout of their machines and of the specific applications resident on their machines.

It is important that each group recognize the need for a LSA and appoint a person who has an adequate skill set to fulfill the requirements of the job. There is often a wildly divergent range of knowledge of computers between groups. The role of each LSA depends upon the computational needs of their respective group.

For example, a group whose research interest is some facet of computer research (graphics, file systems, AI, etc.), and which possess constantly evolving machines, will require a level of system administration which is extremely expert, as the user group is expert. If the group is large and has extensive administrative needs, the LSA may actually be a *group* of full time system administrators.

On the other hand, a group whose interest does not require modified systems, such as a documentation group, will generally utilize stable turnkey systems and will be populated by a user community with basic system skills. The LSA, in this instance, will be called upon to know more about the application software than the computer systems. Indeed, computer system support may require as little as 5% of the LSA's time, while local application support may consume a larger amount of time.

The point is that both groups require a unique level of service based on the type of work within the group, and the needs of both groups require adequate representation within the computing community.

In general, LSAs are the first line of user support. The LSA will be on site to answer user's questions in real time. The LSA can spot developing problems and respond immediately.

The LSA should also serve as the liaison between a group and Central Services. The LSA's representation of the local group's interests in all areas such as policy development, dispute arbitration and allocation of resources, is essential. The LSA is responsible for informing the group of current trends and changes to the computing environment.

Finally, the LSA may be an employee of the local group or an employee of central services assigned to the group. Many groups prefer to have total control (i.e., hire and fire rights). Details can vary from company to company.

Co-operation and Communication are the Key

What is there to prevent confusion, disorder, replication of work and a great bloody mess? Co-operation and communication between the LSA and CS.

First, there must be recognition of one goal: both CS and the LSA exist to provide the best system administrative support services. The feeling of unity and mutual respect must be fostered and supported. To that end, every opportunity for communication must be utilized. Mailing lists are essential. Monthly meetings with the user community are very helpful. Weekly meetings with the system administrators are necessary. Bboards can announce information to users and when archived away, can provide a helpful record of events. Listen to good ideas – they can come from anywhere. Co-development of initiatives invite unity and understanding between groups. Initiatives on policies, security, and standards for the company may lead to unexpected benefits for the whole company. Organize working groups to come to agreement on company wide standards. Promote the sharing of compiled binaries and other resources.

As I am a UNIX system administrator, this paper is slanted by that perspective. Yet, the proposed model can be readily applied to a very heterogeneous site. Elements such as NFS are now commonly utilized by PCs, UNIX based environments (e.g., MACH, AIX, HP-UX), VMS, Macs, 386 architectures, etc. This diversity necessitates a group of communicating individuals, each with a specific expertise.

In a nutshell, I have proposed a system in which there is recognition of the job designation Local System Administrator for each "group" of machines. The LSA is responsible for the efficient adaptation of the machines to the computational task of the group. The Central Services group provides support for those services which are in common use across the computing environment, and also provides information and support services to the LSA.

Experimentation at BBN

Up to this point this has been a theory paper; I did not wake up one morning with all this clearly in my mind. At BBN I soon realized standard methods of centralized UNIX system administration could never work due to administrative boundaries. Searching for alternatives, I reviewed a large body of system administration materials. The books and tutorials on UNIX system administration were based on the premise of centralized control of systems, and did not touch on the topic of distributed system administration responsibilities[1, 2, 3, 4, 5, 6] Indeed, many of the papers which have been presented at past LISA conferences are on tools which enable centralized control over a variety of architectures, and/or a large number of machines.[7, 8, 9] For example, see John Sellens' *Software Maintenance in a Campus Environment: The Xhier Approach* [10] or Bob Arnold's *If You've Seen One UNIX, You've Seen Them All* [11] for good solutions to centralized system administration problems. The only two recent papers on UNIX policy development and implementation assume a selected group has sole responsibilities for system administration.[12, 13] Hence, the development of this model. I will continue this discussion citing examples from BBN. All inaccuracies are mine alone and should not cast a shadow of doubt on the fine quality of support services at BBN.

The group of which I am a member, Distributed Systems and Services (DSS), is chartered to provide UNIX support on a contractual basis to groups within BBN. In addition, the DSS provides e-mail and printing support to the BBN computing environment. The DSS supports roughly 60% of all UNIX based systems at BBN. Some groups which are not supported by the DSS, have their own full time system administrative staff, while others have a part time system administrator, and still other groups do not have any support at all. The divisions buy their machines and exercise primary control over them.

The DSS had discovered that quality support was increasingly difficult to provide to the larger groups, basically due to geographic separations, the requirement for immediate responsiveness, unique configurations and the expanding demands of an expert user group.

In the LSA model cited above, the LSA could be a member of the local group, or a person from the central services group stationed at the local group site. In fact, the DSS stationed two representatives as LSAs in a software development group on a trial basis about a year ago. The group had extensive system administrative requirements which necessitated at least two full time system support staff on site. By all accounts the experiment has worked out very well.

There are more LSA assignments planned. The DSS has a unique LSA assignment plan to facilitate communication and coordination: DSS's LSAs are scheduled to work 4 days a week at the local site. The remaining day is spent with the DSS working on projects. The members of DSS who are assigned "remote posts" are also required to attend the weekly meeting of the DSS. With this representation, all DSS discussions include input from local system administrators.

To facilitate contact between all system administrators and users, the BBN UNIX User Group meetings were initiated. Here, topics of interest to the BBN computing community are addressed along with announcements of changes of service, bug reports, information distribution, etc. A bboard has been dedicated for announcements from DSS and comment by the user community. The user community is encouraged to contribute to enhancement of the BBN computing environment via bbn-public. (See *bbn-public – Contributions from the User Community,* these proceedings). Company standards have been developed by representatives from all areas of BBN. For example, the user community and local system administrators have called for a BBN-wide computer security policy, which is currently under development with representatives from each division.

BBN's path guidelines further illustrate the concept of distributed responsibility. These guidelines were developed by a coalition of local system administrators, users and members of the DSS, headed by Ms. Pam Andrews. The directories /usr/local/{bin,lib,etc,src} are under the control of the local system administrator. /usr/local/pub contains random freeware contributed and supported by members of the BBN computing community: bbn-public. /usr/local/<project> contains project specific items which are managed by the local system administrator in conjunction with the specific project. /usr/local/bbn contains packages of common interest to the BBN computing environment and is managed by the DSS. As X and gnu packages are **large** and of common interest they are segmented into /usr/local/X11R5 and /usr/local/gnu and are managed by the DSS. Announcements by the DSS to the BBN computing community were made advising the users of these path changes.

The DSS is constantly identifying services which are of interest to the total BBN computing community, and striving to fulfill these services. One very successful venture has been the X file server project headed by Ms. Peifong Ren. Distributed throughout the company are sparc systems which export the necessary X11R5 binaries to any UNIX based system at BBN. Hence, unless a local group has need of some very unusual configuration of X, the local system administrator is free from the X ball and chain. There are other examples too numerous to mention, but for the future, there is only improvement.

Conclusion

In the course of writing this paper, I found myself using the terms "control" and "support" almost interchangeably. This is the crux of the problem. How can an organization support a machine if it does not control it? In general, groups want their own machines configured and supported to fit **their** needs, and are not concerned with the management concerns of other groups. While centralized control and support may work and continue to work well at some sites, it is not the answer for all sites.

I have proposed a model of system administration where the establishment of the position of the local system administrator is essential. I advocate a strong Local System Administrator who is ultimately responsible for the efficient operation of a group's machines. Communication between Central Services and the Local System Administrator is fundamental to quality system support.

While computing technology has become increasingly distributed, UNIX system administration has lagged behind. Hopefully this model will provide some insight into this problem and will facilitate the search for solutions.

Acknowledgments

Much of the thought which resulted in this paper is due to the unique challenges presented to me at my position at BBN. Many thanks go to my manager Frank Corcoran, who is willing to argue with me :-); and my co-workers, the members of DSS: Pat Harmon, Betty O'Neil, Pei Ren, Pam Andrews, Ed Eng, Frank Lonigro, John Orethoefer and David Nye and the members of the BBN computing community. I am surprised and delighted that they listen to me.

References

1. B. H. Hunter and K. B. Hunter, *UNIX Systems: Advanced Administration and Management Handbook*, Macmillan Publishing Co., New York NY, 1991.
2. D. Fiedler and B. H. Hunter, *UNIX System Administration*, Hayden Books, Indianapolis, 1986.
3. A. Frich, *Essential System Administration*, O'Reilly & Associates, Inc., Sebastopol CA, 1991.
4. E. Nemeth, G. Snyder, and S. Seebass, *UNIX System Administration Handbook*, Prentice Hall, Englewood Cliffs NJ, 1989.
5. B. Chapman, T. Christiansen, T. Hein, R. Kolstad, H. Morreale , E. Nemeth , and J. Polk, "Advanced Topics in UNIX System Administration ," *Course notes, LISA V*, 1991.
6. E. Nemeth and R. Kolstad, "Advaced Topics in UNIX System Administration," *Course notes, USENIX summer conference*, 1991.
7. *Proc. USENIX Workshop on Large Installation Systems Administration III*, Austin TX, Sept. 7-8, 1989.
8. *Proc. USENIX Conf. on Large Installation Systems Administration IV*, Colorado Springs CO, Oct. 18-19, 1990.
9. *Proc. USENIX Conf. on Large Installation Systems Administration V*, San Diego CA, Sept. 30-Oct. 3, 1991.
10. J. Sellens, "Software Maintenance in a Campus Environment: The Xhier Approach," *Proc. USENIX LISA V*, pp. 21-28, San Diego CA, Sept. 30-Oct. 3, 1991.
11. B. Arnold, "If You've Seen One UNIX, You've Seen Them All," *Proc. USENIX LISA V*, pp. 11-19, San Diego CA, Sept. 30-Oct. 3, 1991.
12. E. D. Zwicky, S. Simmons, and R. Dalton, "Policy as a System Administration Tool," *Proc. USENIX Workshop on Large Installation Systems Administration IV*, pp. 115-123, Colorado Springs CO, Oct. 18-19, 1990.
13. B. Howell and B. Satdeva, "We Have Met the Enemy, An Informal Survey of Policy Practices in the Internetworked Community," *Proc. USENIX LISA V*, pp. 159-170, San Diego CA, Sept. 30-Oct. 3, 1991.

Author Information

Peg Schafer started her career in computing by feeding card readers in 1972. After graduating with a BFA in sculpture, she financed her wood working tools by working at the department of Computer Science and Robotics at CMU. She migrated to Bellcore where she learned the finer points of machine room construction. Eventually, her friends gave her a loom and shipped her off to graduate school at the Media Lab at MIT. Now, master's degree in hand, freed from the bondage of graduate school, Peg's title is Senior Systems Programmer for the Distributed Systems and Services group at Bolt Beranek and Newman Inc. Peg and her husband

David Zeltzer are currently looking for a new home to house the new loom Peg is going to buy. She can be reached by snail mail: 10 Moulton Street, Cambridge MA 02138, 617-873-2626 or by email at peg@bbn.com. The opinions expressed here are Peg Schafer's, not BBN's. If you wish to ask questions please feel free to send e-mail.

Appendix – How to Start?

If you think there are some useful ideas in this model, and you may wish to implement them, you first must understand that changes will take a long time. Here are some hopefully helpful suggestions.

Identify what you believe to be the major problems. Look at the structure of the corporation. Is it disorganized? How are the funding and other resources distributed? Remember, all managers are concerned with funding issues, so any changes will have to address that issue. Consider the "culture" of the company. Consider past historical occurrences. What are the current methods of work? How have they evolved into the current practices? Consider distributed computing and its future trends. Does your company have a plan to best utilize these trends? Consider the computational needs of each group. Identify key and influential individuals in the corporate structure. Identify key and influential individuals in the user community. Get out of your chair and go and speak with them. Have something interesting to say, and ask questions. After speaking with the people, re-evaluate what is the major problem. Pick a small part of the problem to fix. What are the weaknesses? What are the strengths? Dispell the myth that "users" are useless. Get to know the users. Get the users "on your side". Form user groups so they may express an opinion on the direction of computer support issues.

By the time policies are completed, they are prime candidates for revision. Modify existing policies to include the responsibilities of the local system administrator. Centralized policies are becoming broader and more generalized. To be effective, they outline "principles" then break down into specialized areas. Policies, if advertised as "documented methods of work," are less threatening, easier to sell, and really, more to the point.

Customer Satisfaction Metrics and Measurement

Carol Kubicki – Motorola Cellular Infrastructure Group

ABSTRACT

The thought of launching a customer satisfaction program can cause anxiety for a system administrator. However, increasingly quality conscious user populations and a trend toward revenue generating charge backs for overhead services can force the system administrator into monitoring and reporting the quality of services provided to a population. These reporting efforts might be undertaken by the industrious administrator without the pressure of external forces because the products of an on-going customer satisfaction metrics and measurement program can be very beneficial to any group struggling to determine where to most effectively apply its limited resources.

The critical elements of a satisfaction program include more than machine uptime statistics. Customer focus groups are used to gather data to feed a satisfaction survey. The mechanics of creating such a program including some potential problems to be aware of are outlined.

Introduction

Motorola launched a corporation wide program to reach a goal of Six Sigma Quality[1] in all business units in the 1980's. In 1988, Motorola was honored with the Malcolm Baldrige National Quality Award. This honor, coupled with ambitious goals has applied pressure for all departments to demonstrate their quality and satisfaction programs.

Quality and satisfaction measurement are well documented disciplines in the Manufacturing and Software Engineering worlds but these disciplines don't map easily to System Administration. Defects per million opportunities, thousands of lines of code, and the impact of competition are difficult to parallel for the system administrator seeking to benchmark and improve upon quality and satisfaction levels.

Uptime Isn't enough

Many administrators now publish uptime or downtime statistics on their major systems (if not all systems). In the mainframe tradition, one can state that they have achieved *x%* uptime for a given period. However, uptime is not the only factor relevant to customer satisfaction. It is agreed that it can be a major factor, but system performance and the availability of any network licensed or distributed tools can be more important when considering the customer perspective.

The determination of factors relevant to customer satisfaction can best be done by the customer. Although our customers are not likely to routinely expound on the benefits of some of our most fundamental responsibilities such as disaster recovery and security, they can identify the things that are most important to them. These factors are termed customer perceived factors. Complementing the customer perceived factors are the infrastructural factors such as the availability and performance of network segments.

Customer Perceived Factors

The customer perceived factors are those things that the customer is likely to notice and remark about. These many subjective factors will vary between and among different sites. Some populations are obsessed with performance issues, others are concerned that the most harmless error message is signaling the end of the world. In discussing the provisions of connectivity, uptime, security and integrity with the population served at a site, an understanding is gained of what is most important to them - in their terms. Once these criteria are identified, the performance of the organization and it's processes used in meeting (and exceeding) the expectations of the customers can be monitored, measured and reported. Even more important than the act of measuring, is that the measurements are repeated and used to focus improvement efforts. Don't measure just to measure.

Infrastructural Factors

Much less subjective in nature are the infrastructural factors of the campus network. Measurements of degradation and collisions are infrastructural and unquestionably important to the site as a whole. However, it is rare for a customer to remark specifically about an infrastructural element. In considering the perceptions of the customer, it must be assumed that the infrastructure is of high quality. Objective processes must be in place to monitor and

[1] Statistically, a six sigma quality level means that defects will occur only 3.4 times for every one million opportunities to create defects.

measure the quality of the infrastructure. Should a problem be detected, it must be immediately resolved to ensure that it will not become a customer perceived problem. This assertion is a great justification for pro-active network management.

Many third party products are available to assist in the network management effort. Pro-active network management can be accomplished via in-house scripts or the most expensive third party solutions. The point is, from the administrators perspective, the customer perceived factors become important only after the infrastructural areas are addressed. The first step in this process is to identify infrastructural elements and ensure that their performance is monitored, reported and improved.

Customer Communication

Customer communication sessions will focus on the customer perceived factors. These factors will be defined by the customer groups. In addition to being an initial step in the survey process, these communication sessions can encourage interaction between administrators and customers that isn't motivated by a problem or an emergency. Too often, administrators believe that customers think of them only during a crisis. The satisfaction program will prompt the customers to think of the many different things the administrators do, and at the same time, the administrators will see the customers for what they are - *customers*.

The Topic Guide

Key to the discussions to be held with the customer population is the Topic Guide. This guide will serve as the open-ended agenda or outline for several different meetings with varied representatives of the customer base. The use of this guide will ensure that all focus groups stick to the same agenda, and they at least begin to cover the same material. The Topic Guide is used in small group sessions with the customers to begin to narrow the focus from all services provided, to only those that are important to the customers.

The Topic Guide is a concise outline of the products and services provided to the customers. The outline is created internally by all facets of the organization. In creating the Topic Guide, all of the different products and services should be considered, as well as, the different customers to whom these products and services are provided. Often, these different customer groups are diverse in terms of geography, skill level, and the products or services that they require.

The Focus Groups

Diversity is good and should be sought after in the customer focus groups. Schedule different types of individuals to participate at the same session. It is not necessary to meet with all customers, but a good representation of the population is critical. Strive for the participation of members of the technical, administrative, and management staffs at each session. As each of these groups raise their own issues, they will be discussed within the group, as a group they will determine which issues are most important.

Listening

Look within the organization to identify the individual with the best active listening skills. The person who is to catalyze the focus groups must also be fully aware of all of the products and services provided by the organization to allow them to re-focus the discussions as necessary. It is critical that the catalyst does not attempt to justify or defend the organization during the focus group sessions. No free exchange of ideas can come out of a confrontation. The important things to listen for are:

- What is important to the customer?
- Why is it important to the customer?
- How can we help the customer do their job better?

The Session

The Topic Guide is provided to the focus group participants in advance with an explanation of what it is, and what it is for. While the catalyst must be a good listener, and be able to re-focus the session as needed, this person must also be able to record all of the issues raised at the session. An actual tape recorder might be a good solution, but it could be intimidating to the customer group. Depending on the climate, pen and paper might be the best solution. All of the elements in the Topic Guide are addressed in the session. The customer group will determine how important these items are and explicitly state their importance, or more subtlely indicate the level of importance by the amount of time spent on a particular subject, or by the number of questions or problems that they associate with a particular topic. The session catalyst is tasked with making the transcript of the session a meaningful set of important products, services and concerns. Obviously all of the sessions should be conducted in the same manner and if possible, by the same person to ensure that the final product is consistent across all focus sessions.

The Product

Out of the Topic Guide sessions will come the product of the notes taken by the person who conducted the sessions. This product should be a unified document outlining the importance of various products and services and the concerns expressed by the participants. This unified document will be used to identify the subjects of the survey questions. The value of this work is clear. Given the list of things that are important to the customer, the scope of the survey is limited to a only those things.

The Survey

The actual survey can be conducted using whatever mechanism makes sense to contact the customer population. A hard-copy survey offers the advantage of anonymity, but lacks features such as the automatic calculation of results. An e-mail survey might provide on-line data, but can cause the customers concern because the results are not anonymous. A survey program of some type can be devised to collect and tabulate data as anonymously as required. The mechanism used to conduct the survey is usually determined by how much work one expects to put into the tabulation of the results. The amount of work can be estimated based on the quantity and complexity of the questions, as well as, the number of expected participants.

The Questions

The questions should focus on items identified as important in the topic guide discussions. The customer groups who participated in the Topic Guide sessions can be useful in reviewing the questions for clarity. Questions should be designed to include both the respondents current satisfaction level with a particular product or service, and the overall importance of that product or service. In addition, questions should seek to determine why a customer is satisfied or dissatisfied with the product or service. Such questions are vital because following the survey, they can indicate where focused improvements should be made.

Quantifiable

The satisfaction and importance levels can be actual numbers (for example, on a scale from 1-5) or can be related to numbers using a scale from poor to excellent. For repeatability purposes, almost every question must be quantifiable. The few that are not quantifiable can be considered comments.

Repeatability

The survey should be designed to be conducted again and again. Ideally, one would measure for a bench mark, launch a program of improvements (as prescribed by the survey data), and then measure again to ensure that the improvements have been properly targeted and to determine the pay-back. The entire process can be repeated to continue improvement identification.

Comments

Comments can be both an asset and a liability. Some of the most valuable tangible information can come from survey comments, however often participants request some type of action in their comments. These comments must be addressed in some way. If they are ignored, the participants may lose faith in the survey process and might not participate in the future. When asking for comments, the liability of responding to them is accepted.

Customer Validation

The results of the survey are more meaningful both internally, to the the administrative group, and externally to the customers, if the survey is a customer validated survey. To validate the survey, representatives of the customer population participate in the design of the survey through their participation in the Topic Guide sessions and also in their review of the initial survey questions. Another critical element in the validation of the survey is the publication of the results, methodology, and any plans to act on the survey results.

Major Factors Contributing to Satisfaction Levels

In addition the network infrastructure factors customer perceived factors such as the following have been shown to impact customer satisfaction levels at a particular site. As the survey process is used at different sites, with different customer populations, different customer perceived factors might be identified. The importance rating assigned to each of these factors is expected to vary as well.

System Availability

One of the most obvious services administrators provide is system uptime. After all, other customer concerns such as performance and response time are non-issues if the machines are not up or usable. System availability can be considered to be partially an infrastructural factor, and partially a customer perceived factor. The customer is not always aware of what specific network service or data resides on an affected server. Downtime statistics should be posted for all major systems. Analysis of the downtime trends can shed light on satisfaction levels, as well as, major problems.

Scheduled downtime versus unscheduled downtime (and/or prime-time versus non-prime-time) should be broken out if possible (perhaps a stacked bar) because these different types of downtime have different affects on the satisfaction levels. Survey and/or Topic Guide data will dictate what downtime must be tracked and how it should be reported.

System Response Time

Often a customer will report "The network is down" when a machine is simply slow. The customer perception is that, at that time, the network is not usable. Indeed from a customer's point of view, the network very well could be down. This perception might be caused by a congested network segment, or a heavily loaded server. Even though the particular segment or server that is needed is operational, it might be slow, causing the customer to perceive that there is a problem.

Acceptable performance levels for network segments and servers can be drafted in a Performance Agreement. Current performance should be measured and analyzed to identify improvement opportun-

ities. Constant measurement will help to identify the most effective means to improve response time.

Error Messages

Realizing that many customers work with workstation consoles on their desks we can see that many of these customers are subjected to various messages that are reported to these consoles. Often simply informative messages are interpreted by the customers to be error messages. When the customer perceives an error condition, the level of satisfaction with the services provided drops.

The various messages reported should be examined. For example, factors causing messages such as "server x not responding still trying" should be identified. Messages such as these should be virtually eliminated. Indeed such messages will be generated when a workstation is making attempts to access a server and can't because it is slow or down. However, efforts made to decrease the number of servers used by any given user, increase the performance of the servers (in terms of speed), and keeping the machines up will decrease the frequency of such error messages.

Availability of Tools

Often workstations and servers are available and fully functioning, but major applications are not available to the customer. There might be a problem with a license server or database. Information such as how often these conditions occur and the duration of such outages could be used to increase the availability of these resources or target them more effectively. It should be possible to report the "down time" for major applications much like that of actual hosts.

Customer Interactions

Customers might not deal with all members of the system administration staff. Their interactions may be limited to the Help Desk staff, a focal point administrator, or the Field Service providers (or any other organization contracted to assist in support). These interactions can affect satisfaction levels. Therefore, the satisfaction levels associated with the services provided by these front line contacts must be included in the survey as well. In the best case, databases are used to track help desk and field service activity to monitor the type of activity, severity levels, and cycle-time.

Help Desk

Ideally in terms of monitoring and measuring, every system administration group uses a help desk, or a problem logging tool to centrally log and monitor all customer reported problems. Many different third party application are available to assist in this effort.

Failure Analysis

A quick analysis of the various problem reports flowing through one help desk revealed that there are essentially three types of problem reports. Some are indeed service affecting problems requiring resolution, but others are simply questions or service requests. Although it is true that in the purest sense one could consider all customer reported problems to be failures (for example a failure in training), it is more practical to consider only those problems that the organization can resolve directly as failures. It is therefore very important that the calls to the help desk be easily classified as failures, questions or service requests. This data must be made easily available for report generation.

One simple system that seems to work well uses a matrix to classify reported problems, see Table 1.

Through the use of this matrix, problem reports can be classified as failures, questions or requests in hardware, software, or service. All problem reports can be classified in one of these categories. Through the use of such a classification system, more valuable data detailing the quantity of different types of failures is available. Using such information, improvements can be more specifically targeted to problem areas.

Even more data can be made available if the matrix is expanded to include a classification for internal and external failures. The administrative group can then focus directly on the internal failures. Once these internal failures are isolated, they can be examined via Pareto analysis to both identify the most effective improvement strategies and predict the results of the implementation of those strategies.

	Failure	Question	Request
Hardware	Hardware Failure	Available Resources	Purchase Consultation Install/Move
Software	Software Failure Admin Error	User Error Usage Question	Tool Install Tool Consultation
Service	Delinquent Request Full Filesystem	Problem on another net	Restore Request Account Request

Table 1: Classification Matrix

Cycle Time

Response time (by definition of severity level) to customer calls is considered important both to the customers and internally to the administrators. It is therefore necessary to monitor response time to reported problems. Obviously, the satisfied customer has no problems, but if a problem does occur it should be resolved promptly (delays will lead to further dissatisfaction). Response time data must be automatically generated and easily available to report generators.

Cycle time as a function of severity level

Response time can be broken out by severity level. In the best implementations, cycle time is monitored as the customer would monitor it. Specifically, if the customer expectation is to work 24 hours a day to resolve a problem, then that is how the cycle time is reported (a 24 hour clock is used for the most critical problems). If minor problems can be addressed during business hours as available, then the cycle time for these less critical problems is measured on a "business hours clock".

Field Service Providers

Field service providers or any other organizations under contract that might come into contact with your customers on your behalf should be held to the same standards as your own organization. Monitoring and measuring the performance of these service providers can pay off tremendously during contract negotiations.

Specifically useful measurements include the number of visits required to resolve a reported problem. Often the customer is subjected to repeated service interruptions because of a problem the field service personnel might have in diagnosing a hardware problem. The customer becomes more dissatisfied with the administration group who is ultimately responsible.

Problems To Watch For

Customer Expectation

Those who participate in the survey process will immediately expect improvements. Launching such a program requires commitment to follow through with an improvement plan and results.

Managements Use of the Numbers

It is especially important that the entire administration group buy into this measurement process without fear. The buy in is required to make extra and honest efforts in terms of recording or entering data to be tracked for analysis. It should be acknowledged that not every facet of our efforts can be shown with these infrastructural and customer perceived measurements. It can be somewhat degrading to have all of the activities of the "super users" be reduced to an endless queue of problem reports. Work quotas and individualized rewards and penalties based on the quality and satisfaction information should be discouraged.

Management will want to utilize any data for manpower justifications or adjustments. This should only be an option if truly warranted by the data. Careful and complete descriptions of any data published are required to ensure that it is not mis-used or mis-interpreted.

Survey Data/Process

The survey must be designed with the intention to quantify and repeat each element. All efforts should be made to make improvements, nothing should be done just for the sake of doing it. Going through the motions and creating pretty quality charts won't do any good if the information is never used to make improvements.

Action of some type is required when requested during the Topic Guide sessions or in the survey comments.

Conclusion

Conducting a satisfaction survey to identify and constantly monitor the customer perceived factors provides beneficial information. The data can assist the system administrator by identifying common and reoccurring problem areas. With focused efforts, problems can be reduced if not eliminated to free the administrators time for more challenging project work.

Acknowledgements

Many people have been involved in the design and execution of the Customer Satisfaction Program discussed in this paper. Olga Striltschuk of Product Support Quality Assurance provided early guidance. Sam Falkner (now with SunSoft in Colorado Springs) provided many automated tools that allowed the work to progress and continue. Kris Bennett directed the project and provided valuable feedback.

Author Information

Carol Kubicki completed her undergraduate studies at the University of Rochester, Rochester, New York where she was employed as an Operating Systems Programmer in the Unix Group at the University Computing Center. She is currently at Motorola's Cellular Infrastructure Group in Arlington Heights, Illinois employed as a Network and Systems Engineer. She is working toward a Masters in Management and Organizational Behavior at Illinois Benedictine College. Reach her via U.S. Mail at Motorola; 1501 West Shure Drive; Arlington Heights, IL 60004. Reach her electronically at kubicki@mot.com.

References

[1] Bennett, K. & Kubicki, C. SSG Customer Satisfaction Survey Results Information. *EMX Filememo 1545*. Motorola, Arlington Heights, Illinois, 1990

[2] Gehred, J. & Hinz, D. RTSG Network Management Metrics. Internal document of Information Technology Services, Motorola, Arlington Heights, Illinois, 1991.

[3] Kubicki, C. Proposed Customer Satisfaction Metrics. Internal document of Information Technology Services, Motorola, Arlington Heights, Illinois, 1991.

[4] Striltschuk, O. Customer Satisfaction Program. Internal document of Product Support Quality Assurance, Motorola, Arlington Heights, Illinois, 1990.

[5] Von Mayrhauser, A. *Software Engineering Methods and Management*, Academic Press, San Diego, CA 1990.

Request: A Tool for Training New Sys Admins and Managing Old Ones

James M. Sharp – Lawrence Livermore National Laboratory

ABSTRACT

It seems that one of the greatest problems facing senior system administrators today is how to train new system administrators. There never seems to be enough time in a day to get done what needed to be done last week, let alone find extra time to train a new sys admin. And if you were actually willing to give up some time to train, how would you go about doing it? Do you start with adding users or adding hosts?

This paper will not only define an effective way to train new system administrators, but also show a practical way to manage tasks, users and administrators through the use of a simple tool called *request.*

The Problem

In today's market, good sys admin are hard to come by. It seems like if you really want a good and reliable sys admin, you need to train them yourself. This leads to a second problem, most senior sys admins are way over worked due to the demand verses availability of good sys admin. The result is what is seen most often in large sites: One or two really good people doing all the work because they know how and a group of juniors trying to find things to do to keep busy. Everyone is frustrated and burn out is always just around the corner. What we need is a simple method for training that is not overly time consuming. It would also be helpful if this method or tool could help you manage your time and your staff's time.

History

I'm part of a team of nine sys admins that manages the largest group of networks at LLNL. We have seven different networks running through 25 different trailers and buildings. We have a total of 350 UNIX workstations and 500+ Mac and PC's. We started out with just three senior sys admins for the entire network and added juniors as we grew. Our training technique was kind of like farming, we sectioned up the networks and gave each person a parcel that they were responsible to maintain. Our philosophy was "sink or swim." Whatever the users needed on your part of the net was your responsibility to take care of. This included everything from adding accounts to installing and configuring workstations. As senior sys admin, we figured that if a junior needed help they'd ask and if we didn't hear from them or the users then all must be well.

This philosophy/technique seemed to be working just fine until we had a major incident where we lost almost five months of project critical data. After the dust settled from this incident. We decided that we needed a different training and overall network management philosophy. Out of the rubble and user distrust, *request* was born.

The Solution

We decided that the solution needed to begin with a new training philosophy. We came up with the layered model. Instead of assigning each sys admin a parcel of machines and users on the network that would require the full range of skills, we would treat the network as one big farm and as jobs were required they would be assigned out according to skill level. This way if a sys admin only had a junior set of skills then he would only be assigned jobs that would accent his skill set. Figure 1 is a diagram of the layered model.

Advanced tasks requiring Advanced skills
Moderate tasks requiring Moderate skills
Easy tasks requiring Beginner skills

Figure 1: System admin task layers

As a sys admin's skill set increased so could the complexity of the tasks assigned to him. By using this method, a sys admin's skill's could be increased at a somewhat controllable rate while still accomplishing all the tasks required by the network. As questions began to subside at a particular level, you then knew that the sys admin was ready to move up.

Implementation

With a philosophy agreed upon, the next step was the development of an implementation plan. I dug up an old hotline call tracking program and began modifying it to meet our current needs. What resulted was a set of shell scripts that allowed user generated requests to be logged from any network workstation. These requests were then assigned to sys admins according to skill sets. When the job was completed, the user and senior sys admins received a mail message with closing comments.

```
% request
Hello James M. Sharp.
  Welcome to the O-Div User Request
  System, Rel 2.0 If you have any
  questions or problems please send
  mail to:  request_mgr@s1.gov
Please type a one line summary of
your request:
Please install 2.0GB disk on atlantis
Would you like to include a more
detailed description? (y/n)[y]: n
Please type the date by which
resolution is required
(mo/da/yr): 11/1/92
Who would you like to carbon copy
on this request?
Press RETURN when done.
cc: tmd mmy
```

Figure 2: Help request session

```
   O-Div User Request #1109
1. Originator:  James M. Sharp
2. Username:    jms
3. cc:          tmd mmy
4. Abstract:    Please install new
                2.0GB disk on atlantis
5. Required by: 11/01/92
6. Edit detailed description
Would you like to make any corrections
to the above data? (y/n)[n]: n
Would you like to submit this request?
(y/n)[y]: y
  Your request has been logged as
  O-Div User Request Number 1109.
  Please use this number when
  referencing this O-Div User
  Request in the future.  Someone
  from the O-Div Network Support
  Team will get back to you as soon
  as possible.  Thank you for
  entering your request.
```

Figure 3: Request editing

More and more features were added and eventually awk, sed and csh were not enough to keep up with the volume of requests that were being processed. Over the last eight months we have completely re-designed *request*. The newest version now runs under perl and uses a dbm file to store the request information. The user interface is still curses based, but can now be filled out in a matter of moments. Figure 2 shows the input form of *request* while Figure 3 shows the verification form.

One of the design issues that we faced was how to determine the severity of a request. In the first version we had the users pick between *critical, urgent, need* and *minor*. This seemed to work except for those cases in which the severity of a request needed to be upgraded. A request that was due a month from now would probably be *minor* or a *need,* however, that same request a month later might be *critical*. Users also complained that in general it was difficult to determine the severity of their own requests. So we eliminated the severity field all together and now severity is based solely on the date the request is due. Simply, requests that are due today are more critical than requests that are due tomorrow.

Along with an easy to use user interface, *request* also provides a number of reports that we find very useful in managing both sys admins and the network. Figure 4 shows a list of open requests for jms.

Each sys admin can view his own list and use the information to prioritize his time. We also use a report similar to this one in which all open requests for each sys admin are listed. This report is used in our Network Operations Meeting. We meet weekly to discuss trouble spots on the network and to offer each other insight on some of the more difficult requests.

Another useful feature of *request* is the automated assignment function. When new requests are logged they are automatically assigned to "nobody." Then the sys admin in charge of making assignments can run through the routine shown in Figure 5 to assign the new tasks to particular sys admins.

```
List of open requests for:  jms
 Num Who Date Due  Abstract
1106 jms 08/04/92 Install patches on unclassified machines.
1107 jms 08/06/92 please install mxgdb
1108 jms 09/01/92 Remove mordor from uucp maps.
1109 jms 11/01/92 Please install new 2.0GB disk on atlantis
1111 jms 08/29/92 psroff is not working on nimitz
```

Figure 4: Request Summary

Request can also be viewed by typing "request -v [req#]" as in Figure 6. As you can see all updates to the request are included in the detailed description so that progress can be monitored and tracked.

Application

For Training

Since we began using *request* we have found that new sys admins can be trained by more than one senior sys admin based on the particular task that they are working on. This has been one of the greatest benefits for both new and old sys admin. For new sys admin, *request* provides greater exposure to multiple nets/machines and multiple levels of expertise. For old sys admins, the benefit is that no one person is stuck with all the training duties.

We've also found *request* very useful in grooming sys admins in a particular area. For example, a sys admin who expresses an interest in security then receives all assignments pertaining to security so that he can gain the experience in his area of interest.

```
% request -l nobody
List of open requests for:  nobody

Num   Who    Date Due  Abstract
1120 nobody 09/04/92  Install X11R5 on sentry
1121 nobody 09/17/92  please add account for dave
1122 nobody 10/01/92  yppasswd not working on dixie

% request -a 1120 1121
1120  Abstract: Install X11R5 on sentry
1120 is currently assigned to nobody
Do you want to reassign req 1120?
[y/n]: [n] y

Reassign req 1120 to:  jms

1120 is assigned to: jms

1121  Abstract: please add account for dave
1121 is currently assigned to nobody
Do you want to reassign req 1121?
[y/n]: [n] y

Reassign req 1121 to:  ark

1121 is assigned to: ark
```

Figure 5: Reassignment

```
% request -v 1120

                      O-Div User Request #1121
================================================
State: open                                    Required by: 09/17/92
================================================
Date:          Mon Sep 14 10:25:25 PDT 1992
Originator:    James M. Sharp
Username:      jms
cc:            tmd mmy
Assigned to:   ark
================================================
Abstract:  please add account for dave

Description:

Dave Wright will be starting work on 9/17/92
and will need an account.

ark assigned to request 1121 on Mon Sep 14 12:58:36 PDT 1992
```

Figure 6: Command line request display

For Management of Sys Admins

request has given us the ability to monitor all the tasks being worked on by the sys admin staff. We first established the policy that no one does any work for a user unless there is a related request first. This policy allows the senior sys admin in charge of assigning tasks to see the request first and determine if it is indeed a valid request. The senior sys admin also receives all updates including the closure along with the user who submitted the request.

We've also found task oriented work to be much less stressful on the senior sys admin. They no longer have to feel that they are solely responsible for the entire network. Now they are only responsible for what is on their list like everyone else. It also helps in prioritizing to have each task defined and dated with a due date.

For Management of the Network

As a network manager it is sometimes very difficult to see all the activity on a large network. Many times decisions are made that will solve the problem at hand but in the long run end up a having drastic impact on the network at large. Again with *request* all requested changes to the network are seen by the network manager before they are assigned. This way any potential problems can be defused before they blow up.

The network manager can also use the reports generated by *request* when it comes to determining work load and staffing. You can use *request* to determine how many requests per week/per month that the average sys admin can complete. If that number for your site say is 30 then if you have a volume of 150 requests per month then you would need 5 sys admins to handle the work load. You can also list the closed requests for each sys admin, which includes the date the request was closed. This report is useful in verifying that your average of 30 requests a month are actually being completed by the due dates requested by the users. If requests are not being completed on time then you may need to lower your average number of requests completed per month and also hire more help.

Future Development

I'd like to see a Graphical User Interface added to *request*.

Summary

I personally feel that we are producing more well rounded sys admins than before (and happier ones too). We've also found that our start up time with new sys admins has been cut way down. We add them an account, show them how to use *request,* find out what they know how to do all ready, and start assigning them tasks that meet and slightly exceed their skill set.

Now that we use *request,* our junior sys admins are being trained better; our senior sys admins are better organized; our senior sys admin share in the training duties; we are able to steer sys admins in specialized areas; overall our network runs better; and we function more as an integrated network admin team as opposed to a bunch of individuals. For us, using *request* has been a real success.

Program Availability

request is available on a case by case basis from the author.

Acknowledgments

I would like to thank Shawn Instenes for all his help in converting the request code from csh to perl.

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract no. W-7405-Eng-48.

References

[1] RFC1297, *NOC Internal Integrated Trouble Ticket System; Functional Specification Wishlist*

[2] Wall, Larry and Randal L. Schwartz, *Programming perl.* O'Reilly & Associates, Inc; Sebastopol, CA 1991.

Author Information

James M. Sharp, a Senior Systems Administrator with the Distributed Computing Support Program, has been at Lawrence Livermore National Laboratory since January 1990. Jim has a B.S. in Computer Science and 7 years experience as a UNIX system administrator. Jim is also a member of Bay-LISA, a San Francisco Bay Area user's group for system administrators of large sites. Contact Jim via US Mail at Lawrence Livermore National Laboratory; PO Box 808 L-270; Livermore, CA 94550 or via e-mail at jms@s1.gov.

Typecast: Beyond Cloned Hosts

Elizabeth D. Zwicky – SRI International

ABSTRACT

Typecast is a system for automatically configuring computers according to type definitions. This method allows a la carte installation of local options and automated handling of a wide variety of installation types from within a single program. Typecast is designed to make it possible to modify configurations and add new configurations without changing Typecast itself.

Introduction

If you buy a single computer, you lovingly hand-install it, or at least lovingly hand-modify the installation shipped with it. At about 10 computers, you begin to feel somewhat more sullen about the procedure. The classic next step at this point is to develop a procedure for cloning machines from a master, lovingly hand-installing any personality the machines need afterwards. Even this procedure does not hold out forever, however, as eventually the personalization work becomes a chore. Furthermore, most clone procedures are relatively inflexible; somebody attempting to maintain a loosely affiliated set of facilities is unlikely to be satisfied with a normal cloning procedure.

When SRI's classic cloning procedure became inadequate, I set out to write a language that would allow me to specify a large number of configurations, and customize a vanilla operating system installation appropriately. This way, I could create machines for a variety of different purposes without having to customize each machine myself. This language is called "Typecast" and is built in Perl (in fact, configuration files frequently can contain arbitrary Perl statements). The program reads files in a config directory to determine what types apply to the machine being customized, and then applies the appropriate customizations.

Design Goals

When I started work on this program, we were already using a cloning system written by Dave Curry where machines to be cloned were booted from the network, and then a program was run which formatted the disks, restored dumps of / and /usr onto them, and performed some minimalist customization to correct the host name and network address and set up a few files that changed according to which of our networks the machine was on. This program worked quite well for the purpose it was designed for, which was configuring machines to go on users' desks. The machines in question came in a very small number of configurations, and needed to look as much alike as possible.

Some time after the program was written, however, the wave of machines that came in while we were transitioning from Sun 3s to Sun 4s ended, and the problems we were facing changed. Instead of mostly configuring the same 2 machine types with the same 2 disk types for the same purpose, we are now configuring machines that are all over the map, from MP690s to 3/80s, and that were being used for all sorts of purposes. Most of these machines are not going onto users' desks, but instead are being used for demos, home machines, and project machines. Furthermore, we are increasingly often configuring machines that are not even in our domain.

The new program, therefore, has several goals that the old one didn't. These were:

- Allow machines to be partially configured, as well as completely rebuilt.
- Assume as little as possible about the machine being configured.
- Allow the configuration options to be modified without changing the program code.
- Allow multiple configurations.

At the same time, it shares the old goal of letting unskilled system administrators could do routine installations. Ideally, they should even be able to modify configurations.

Basic Concepts behind Typecast

Typecast consists of a program, which is a parser with almost no knowledge about configuring UNIX machines (it does contain an exclusion list that prevents you from straightforwardly deleting essential files), and a configuration directory structure. The process of configuring a machine is thought of as a process of determining and refining a description of what type of machine it is, and doing things appropriate to that type. The types are represented as a directory tree.

When you run the customize program, it starts at the top of the directory tree, recursively applying type directories. In each type directory, it starts by reading a `types` file. The types file lists possible types and how to figure out which ones apply to the machine that is being configured; each time that the program determines that a type applies, it checks for

a type directory for that type, and applies it if it exists. In each directory, after the program has finished the types file, it runs through a list of files to delete, a list of files to install, a list of files to modify, and a section of arbitrary Perl code.

I chose to put in special support for deleting, installing, and modifying files by looking at the checklists that we maintained for installing machines to local standards by hand. These were the fundamental operations that the checklist relied on. Because they match the intuitive model of configuring a machine, they are easy for unskilled administrators to deal with (although not all of Typecast is).

Determining Types

Possible types and the ways to select them are defined in types files. Types may have arbitrary values or boolean values, and may be specified as exclusive lists. For instance, "hostname" takes an arbitrary value, within limits. "reformat", specifying whether the disks should be reformatted, takes a boolean value. "ypserver" and "ypclient" make an exclusive list. Sub-types can be defined by putting further types files inside the configuration for a type.

The type definition specifies not only whether a type is valued, boolean, or listed, but also how the program will figure out what value to assign to the type. The program can either run a given piece of code to determine the answer, run a piece of code and ask the user to confirm the answer, or simply ask the user. The type file can also specify checks to run on valued types to insure that they are legal (for instance, checking that the hostname is a valid hostname, or allowing only values for "ypdomain" that have matching configuration directories).

Each type definition consists of a type name, followed by a colon and a method for determining the type value. The possible methods are ''ask'', ''guess by'' and ''set by''. (The ''by'' is actually a no-op which serves to remind the human reader that

```
format: ask "Do you want to format sd0?"
restore: ask "Do you want to restore an operating system onto sd0?"
prefix: valued noconfig ask "The / you wish to configure is mounted where? "
hostname: valued guess by test by
chop($hostname = 'hostname');
ENDCODE
/^[a-zA-Z0-9\-\_]+(\.[a-zA-Z0-9\-\_]+)*$/
ENDTEST
hostnumber: valued noconfig guess by test by
if (($name, $aliases, $addrtype, $length, @addrs) = gethostbyname($hostname)){
  ($a, $b, $c, $d) = unpack('C4', $addrs[0]);
  $hostnumber = join ('.', $a, $b, $c, $d);
}
else {
  $hostnumber = "128.18.110.250";
}
ENDCODE
/^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$/
ENDTEST
broadcast: valued noconfig set by
$broadcast = $hostnumber;
$broadcast =~ s/\.[^.]+$/\.255/;
ENDCODE
domain: valued guess by fallback by "."
if ($hostname =~ /[^.]+\.(.+)/){
  $domain = $1;
}
elsif (open(DOMAIN, "</etc/defaultdomain")){
  chop ($domain = <DOMAIN>);
  close DOMAIN;
}
ENDCODE
```

Figure 1: Sample types file

there is attached code, and may be omitted.) The "ask" keyword tells Typecast to ask the user what the value of the variable should be, and takes a double-quoted question as an optional argument. Typecast's default questions are "Is this machine a <type>?" for booleans, and "What is the correct value of <type> for this machine?", which are functional for most types but get dull very fast. The "guess by" keyword tells Typecast to execute the attached code segment, and then ask the user to confirm the resulting value. It does not take an optional question, which tends to lead to stilted little dialogs of the form "Is "dog" the correct value of hostname for this machine?" "no" "What is the correct value of hostname for this machine?". On the other hand, "guess by" leads to intelligent default setting; I find it intensely frustrating to be asked for a hostname and then an Internet address when the computer that's asking has a perfectly good database to look it up in, but I still want to configure hosts that aren't in my nameserver. The "set by" keyword simply sets the variable according to the attached segment without asking for a confirmation.

There are multiple optional keywords that may be added. By default, types are boolean, and if they are true, Typecast attempts to execute a configuration directory with a name that exactly matches the variable name. The "valued" keyword specifies that the type takes an arbitrary value; the "listed" keyword specifies that it takes a value from a given list. The "noconfig" keyword tells Typecast not to look for a matching configuration. There is also a "fallback by" keyword, best explained by example: "domain: valued guess by fallback by "."" with a domain of "wash.erg.sri.com" will cause Typecast to look for a configuration named "wash.erg.sri.com" and if that fails to look for "erg.sri.com", and if that fails to look for "sri.com", and so on.

Finally, the "test by" keyword tells typecast to run the attached test before accepting the given value as a valid value for a type. Generally, this is used for straightforward purposes like making certain that your hostnames don't contain illegal characters and your Internet addresses are dotted quads, but it can be put to more interesting uses. We name servers after 3-letter airport codes, and after several experience with people who made random guesses and named machines after inappropriate or non-existent airports, I am determined to take the text US airport database I have and confirm that servers are genuinely named after airports. (Unfortunately, I don't think I can reasonably ask the configurer to enter the destination latitude and longitude of the computer, so I'll have to show the full airport name and ask for a confirmation, instead of doing something really flashy like saying "You can't name a Kansas computer after a New York airport!")

Figure 1 shows a sample types file.

Files to Delete

File deletions are handled straightforwardly by a file named "delete" which contains a list of full path names. Typecast does have an exception list containing the names of files that it considers unwise to delete (for instance, it refuses to delete "/vmunix" and "/bin/sh"). Wildcards are allowed in delete paths. A sample delete file, from our usual local modifications to a stock SunOS installation:

```
/usr/lib/sendmail.mx
/etc/hostname.*
/etc/defaultdomain
```

Files to Install

Files to install are kept in a subdirectory named "install"; each file's name is the complete path name it will be installed under with slashes changed to backslashes. This produces long filenames that are unpleasant to type to the shell, but it avoids the sparse, deep directory trees that haunt many installation directories. You can see at a glance which files the installation is going to change.

Files may be installed verbatim or with modifications. If the file name begins with an exclamation point, Typecast performs substitutions as follows: for each valued variable "variable", the string "VARIABLE" is replaced by the variable's value, and the string "XXVARIABLEXX" is replaced by "VARIABLE". This crude method for marking variables is derived from an older system we use to maintain printcaps, and should be replaced by one less prone to spurious substitutions, but has worked well so far.

A sample listing of an install directory:

```
\.rhosts
\etc\gateways
\etc\hosts
\etc\hosts.equiv
\etc\imadb_prt
```

```
hostname=HOSTNAME
hostname $hostname

ifconfig lo0 localhost up
ifconfig le0 HOSTNUMBER up arp netmask 255.255.255.0 broadcast BROADCAST
```

Figure 2: Special /etc/rc.boot installation

```
\etc\networks
\etc\ntp.conf
\etc\resolv.conf
!\etc\rc.boot
```

You will note that /etc/rc.boot is installed with modifications; the relevant lines are are shown in Figure 2. HOSTNAME, HOSTNUMBER and BROADCAST are set in the top level types file, as shown above.

Files to Modify

Installation with substitution is used primarily to fill in hostnames, domain names, and network addresses in configuration files. Some files require systematic modification instead of brute force installation, because they contain information that is allowed to vary. For instance, we want some local accounts to remain in password files, while still turning on NIS on machines that are supposed to run it, and starring out passwordless accounts. Similarly,

```
DIRECTORY:/export/root
DIRECTORY:/export/swap
INSIDE
chmod 0755, "$_";
rename("$_", "$_.$domain") unless /\.$domain$/;
ENDINSIDE
FILE:/etc/rc.local
BEFORE
$screenblank = 0;
$domainset = 0;
ENDBEFORE
INSIDE
if (! $domainset && /\/etc\/defaultdomain/) {
# change domain setting from /etc/defaultdomain to explicit
  $_ = "domainname $domain\n";
  $domainset = 1;
}
if (/sendmail/){     # change sendmail from every hour to every 15min
  s/\-q1h/\-q15m/g;
}
if (/ifconfig/){
# ifconfigs belong in rc.boot
  $_ = "# " . $_;
}
if (/^[^#].*ntpd/){   # so we don't put in a second ntpd invocation
  $ntp = 1;
}
if (/rdate/){         # no rdate, ntpd
  $_ = "# " . $_;
}
ENDINSIDE
OUTSIDE
if (! $screenblank){
  print CHANGED
"#
# screen blanker
if [ -f /usr/bin/screenblank ]; then
      screenblank && echo -n ' screenblank'
fi\n\n";
}
ENDOUTSIDE
```

Figure 3: A sample modify file

there are particular entries that we want to add to /etc/rc.local if they are not already there, but we do not want to remove any customizations that are already present.

This is handled through the "modify" file, which contains specifications of file or directory names, followed by Perl code to be executed before a loop, code to be executed within a loop over lines of files or names in a directory, and code to be executed after the loop. (Empty sections may be omitted.) Multiple files or directories may be covered in a single loop.

Figure 3 shows some samples.

Modify files, like types files, tend to have Perl's full measure of obscurity. This is not territory where the untrained should venture. I was unable to come up with a syntax for modify that had enough power to satisfy me and still was highly intuitive. Additional support routines would be helpful (for instance, much of the code shown in the sample could be removed if there was a primitive that said "add this to the end if you don't see this other thing while in the loop").

Procedures to Apply

Anything not caught by the delete-install-modify model can be thrown into a file named "code" which simply contains raw Perl code. Because this code executes in the Typecast parser's environment, it does have access to the question routines that Typecast uses, making it easy to ask questions. The "format" type that we use consists of a code file containing a threatening warning statement and the code shown in Figure 4.

Experiences with the Configuration System

Building the Typecast parser was actually relatively simple. Building the configuration files is another matter. All of our configurations are still under construction, as each new machine adds some new twist. This is partly a function of the fact that we are attacking a very complex problem that we haven't previously attempted to formalize rigorously. The hand procedures consisted of a checklist which an experienced administrator read through, deciding which things to do and how to do them on the fly; experimentation then proved whether or not the built system did everything that it was supposed to. This is not a procedure that can readily be translated to code, even when you are the experienced system administrator who normally does it.

On the other hand, it is a major improvement to be able to do any of the work automatically, and at worst, Typecast produces a more interesting class of problems than hand configuration. For one thing, each machine produces a different problem, instead of the same old routine.

Future Work

Obviously, my current mission is perfecting our Typecast configuration files. I also intend to add support for creating Typecast types automatically from a model, and to develop improved primitives for use in modify files. It should be possible to further improve configurability by allowing people to add functions to the delete/install/modify/code quadruplet, but I am not yet convinced that it is worth the effort.

This is a problem which people are just beginning to work on, and I am sure there will be other options available in the future. At the moment, the only other work which I am aware of is Steve Romig's paper from LISA VI entitled "Customizing Cloned Hosts (or Cloning Customized Hosts)". Steve's system is sufficient for the sorts of customizations that occur to machines on people's desks (they have printers or not, they have extra disks or not, they have extra programs started at boot), but it is not readily extensible to the extreme variations we deal with ("Well, I want a standard Sun

```
if (&confirm("Are you absolutely sure that you want to destroy sd0?")){
 (@types) = </usr/facility/clone/data/format.*>;
 grep(s/\/usr\/facility\/clone\/data\/format\.//, @types);
 $type = &asklist(1, "Which of these types is sd0:", @types);
 print "Partitioning drive sd0, using /usr/facility/clone/data/format.$type\n";
 system "format > /tmp/clone.log < /usr/facility/clone/data/format.$type";
 print "Creating root file system\n";
 system "newfs /dev/rsd0a >> /tmp/clone.log";
 print "Creating /usr file system\n";
 system "newfs /dev/rsd0g >> /tmp/clone.log";
}
else {
  print "sd0 will be left alone";
}
```

Figure 4: A confirmer and warning code

configuration, except I want it to be able to work in our environment until I disconnect it and run standalone for a while and then let the client have it.'')

Availability

Typecast is available via ftp on ftp.erg.sri.com in pub/packages/typecast. Documentation is currently non-existent.

Author Information

Elizabeth Zwicky is a senior system administrator for the Information, Telecommunications, and Automation Division at SRI International and interim president of SAGE (the Systems Administration Guild, a USENIX Special Technical Group). In moments of desperation, she regains calm by contemplating the ironies and absurdities of the concepts ''the paperless office'' and ''the computer as labor-saving device''. Reach her via U.S. mail at SRI International; 333 Ravenswood Ave.; Menlo Park, CA 94025. Reach her via electronic mail at zwicky@erg.sri.com.

So Many Workstations, So Little Time

Helen E. Harrison – SAS Institute Inc.

ABSTRACT

In the spring of 1991, SAS Institute Inc. committed to replace its software development network of 400 Apollo workstations with the newly announced HP 700 workstations. This launched a nine month project we have come to call "The Conversion" which culminated in the installation of a major distributed network over a single weekend. During early December, 1991, 300 developers went home on Friday leaving an Apollo workstation on their desk and returned the following Monday to find a brand new HP 9000/720, fully configured for the SAS software development environment. This paper describes the processes involved in accomplishing this feat and what we learned along the way. It concentrates on the system administration and support strategies that were developed to support this new environment. Finally, we present an overview of the structure of the project as a whole, from design committees to the logistics of the installation weekend.

Most large networks start out as small networks and grow, with support strategies evolving at the same time. All too often the administrative solutions do not scale up as well as we would like. We had an unusual opportunity to design distributed support solutions from the ground up. We believe that other sites will benefit from the experiences gained in this experiment in systems administration design, and perhaps find some new ways of looking at old problems.

Introduction

SAS Institute Inc. is a privately owned software company whose primary software development took place on a network of Apollo workstations. In the spring of 1991, the Institute decided to replace the Apollos with a network of new, fast HP 9000/700 series machines. To prepare for the arrival of the new workstations we designed a new TCP/IP based network with sufficient bandwidth to support 1000 50+ MIP computers for 3 years, and a network management tool for keeping up with them. Based on requirements of the software developers we implemented a distributed computing environment based on a combination of AFS and NFS. The UNIX support programmers mapped out the system administration solutions that had to be in place to support this new network. These included user account management, installation management, printing services, mail services, backups, console management, problem management, root access and public tools management. We defined what we wanted in each case and started searching. Some solutions we borrowed, some we bought, and a few we invented ourselves. Our findings in each case are presented.

The conversion to the HP development environment was a project that involved people from many different departments and job functions. The conversion project was organized into committees made up of a mix of development and computer support folks, each responsible for a different area. Some committees mirrored its members real jobs while others were volunteer efforts. There were committees from development departments which were responsible for porting various software tools and the SAS software product itself. The User Environment Committee developed a standard windowing and user environment package with local customizations. The Networking Committee constructed a new TCP/IP network. The User Training Committee organized and taught specialized classes on capabilities of the new HPs. The File System Committee framed the distributed computing environment and chose the distributed file systems. The Systems Administration Committee designed the support strategies. The Logistics Committee organized the actual installations of new machines. This paper will focus on the work of the distributed file system, systems administration and logistics committees.

In the summer of 1991 we received our first 50 seed units which were to be used by the conversion committees in preparing for the conversion. Since SAS software development is based on a large common source, we wanted to replace all Apollos simultaneously to minimize impact on the developers. A test installation of 90 machines was scheduled for September. The major installation of over 300 machines happened over a three day weekend in December, as well as a subsequent installation of 90 machines in January. Smaller installations of approximately 50 machines each continued through 1992. The network now consists of 800 HP9000/720 workstations, 40 HP 9000/750 fileservers with 100 gigabytes of disk space connected through 15 Cisco routers and 70 Cabletron hubs.

Distributed Computing Environment

Our users were accustomed to the highly distributed Apollo Domain computing environment. They had sophisticated expectations about the distributed computing environment that they would use on the new HP network. They expected to see familiar features such as a global namespace across the network, a single user account database and home directory on the network. They wanted a unified view of the SAS software development system. They wanted the HPs to look as much like the Apollos as possible.

The SAS software development environment is based on a unified source structure in which global data structures defining the interface to the core supervisor are shared by all developers. SAS source code is really big: over 4 million lines of C code per release track. We wanted a distributed computing environment which would most closely resemble the Apollos while taking best advantage of the computing resources on a software developer's desktop. Transarc's AFS was the distributed filesystem of choice. AFS was attractive because it provided a global namespace and transparency of volume location. Its local caching of files simplified porting of the source management tools which had to handle local caching on the Apollos. AFS's ability for administrators to replicate volumes and move them around in real time was a significant improvement over static methods used in the previous system. On the Apollo network compute tasks were distributed manually. In new system we were able to take advantage of HP's Task Broker product. Task Broker allows one to submit jobs, such as compiles, to a collection of Task Broker nodes who bid on the job based on their configuration and currently available resources.

A major design requirement for the development network was that a developer should be able to continue working productively even if there were servers down or network outages. The Apollo software development network was networked with Apollo token ring. This network was very fragile; someone stepping on a DQC box could bring the whole network down. The Apollo software development network was interconnected with Apollo token ring. Since the new HP 700s were TCP/IP based we had to install a completely new networking infrastructure along with the new computers. Since these new workstations were very fast (2 HPs transmitting continually could easily saturate a standard Ethernet), we designed a highly subnetted network in which workstations connected to Cisco routers via Cabletron hubs. A typical subnet has between 10 to 20 hosts on it. These intelligent routers and HP servers are interconnected on an FDDI backbone. This new network is much more reliable than the Apollo token ring.

Still, developers should not be completely unable to work if a piece of the network has died. Since AFS depends on continuing communication with a database server, even for files on local volumes or files which are locally cached, we put each user's home directory on his machine under the native UNIX filesystem. This enabled the user to continue to work on files in his playpen, read his mail, etc., regardless of the state of the rest of the network. All machines are available to each other through the AMD automounter with the naming convention of /nfs/hostname. To achieve the global namespace of home directories a person's home directory in the password file contains the full pathname (i.e., /nfs/wheels/local/u/heh).

System Administration Requirements

We were faced with the task of developing highly distributed procedures for supporting a large network with minimal staffing -- while supporting a different existing large network with minimal staffing. We had to identify what was missing from HP-UX for it to be supportable on a large scale. Time and personnel constraints meant that we could do only what was absolutely necessary. A further constraint was that our procedures had to support installation of a major portion of the network in a single weekend. We defined a core set of administrative systems which had to be developed to guarantee success. These included:

- Installation management
- Network topology management
- User account management
- Backups
- Printing
- Problem management
- Console management
- Public tools management
- Super user access

The network of machines was expected to be over 400 machines by the end of 1991 and double that before the end of 1992. Solutions which required "touching" every machine had to be avoided. Our primary concern was to develop solutions which addressed the needs of the HP700s, but we also have a changing collection of approximately 200 other UNIX machines from most every major UNIX manufacturer. We wanted solutions which would help with the heterogeneous system management problem and we wanted to leverage solutions developed for this very large stable base to help in supporting our more fluid UNIX environment. The following is an overview of each of these solutions.

Installation Management

Installation management presented us with an unusual problem. Unlike most configuration maintenance solutions, ours not only had to support configuring machines but it also had to do so with

very minimal initial intervention on our part. We wanted to be able to install such a large number of machines in such a short amount of time that we could not individually customize each one as it was installed. In effect they had to be able to install themselves.

HP's Integration Center service, together with a very clever installation management system, gave us what we wanted. The Integration Center burns in new hardware and customizes it according to customer needs. For us they took a preloaded operating system and made the following changes: set its hostname and IP address, installed `/etc/resolv.conf`, and added the following commands to `/etc/rc`:

```
echo "Starting doit..."
rm -rf /usr/tmp/doit
mkdir /usr/tmp/doit rcp
librarian:/doit/bin/doit \
      /usr/tmp/doit
chmod 500 /usr/tmp/doit/doit
/usr/tmp/doit/doit
```

They also added inventory and property tags. For large installations we send HP a tape with a list of hostnames, IP addresses, room numbers and user names, and they send us a tape with all our information plus serial numbers of all components assigned to each hostname, and, of course, a whole bunch of machines ready to be installed.

The actual installation tool, **doit**, runs each time a machine boots. **doit** applies local customizations to a machine based on directions from a central data base file and an associated tree of data to be installed. Each database entry specifics a revision level, "hostclass", and associated file from the installation tree, as well as which action to perform once that file (or directory) is copied to the target machine. Actions are add software, delete software, or run a command. Optionally, **doit** will reboot a machine after an action. For example, one might want to "add" a new kernel and reboot, or "run" the HP-UX **update** command to install a optional product. Each client machine keeps a record of its current revision level, and uses this to determine which additional actions are necessary to keep itself up to date. **doit** is described in detail in a separate paper. [1]

Hostclasses are a concept used not only in **doit**, but throughout our systems support strategies. Hostclasses, developed at MCNC, [2] are a way of referring to a group of machines by a single symbolic name. For example, if we wanted to specify that a **doit** action take place on the machines `larry`, `moe`, and `curley` we could define a hostclass called `stooges`. Set algebra can be used to manipulate or combine classes (i.e., an action might take place on "`=disney.chars + =stooges`"). The hostclass software consists of a set of C library routines for expanding and resolving hostclasses, a user program, **expandhosts**, and the site specific files containing hostclass definitions. Many of the hostclasses used in support of our development network are generated out of our network topology tool, **netmain**, discussed later.

doit uses a pull model for configuration management which requires the client workstation to initiate the update. Sometimes it is necessary to "push" a command out to all workstations asking them to initiate their **update** "pull". We had our first large scale experience with this when it was time for our first operating system upgrade. To upgrade our network to HP-UX 8.07 we had to distribute approximately 150 megabytes to what was then 640 workstations, partly from our **doit** distribution areas and partly though the standard HP network based software update tools. And as usual, we had to do it in a single weekend! Experience with other tools had taught us that it did not take many HP 700 workstations all accessing a single service at the same time to saturate a server's Ethernet bandwidth. Network wide updates needed to be done in a very controlled manner. First we had to replicate HP's **netdist** servers. (**netdist** is HP's network software installation system.) We found that we had disk space for 7 copies of our **netdist** area. Experimentation showed that each server could handle 11 simultaneous **netdist** updates. This meant that we could upgrade machines 77 at a time. To schedule these upgrades we created a **doit** action which would select a random **netdist** server and then wait until that server was available by monitoring a queue of currently active connections. We started the update process by running a script that rebooted each workstation at one minute intervals. Each machine update took an average of 1 hour. This process was totally automated and took a total of 15.5 hours to upgrade 638 machines.

Public Tools Management

It quickly became clear with our first 25 new machines that we needed a strategy for dealing with all the additional nifty utilities that everyone just has to have. We set up a shared `/usr/local` tree. Encouraging users to write, compile, install and support "local" utilities was a goal. We did not want a system which would require additional work from us to install, but at the same time we needed a system which would maintain order and clearly delineate who supported which utility. We considered writing tools which would manage compiling and installing utilities and associated libraries and manual pages, and ensuring that a program's owners were properly properly identified. As an interim measure we came up with a policy for managing local programs which turned out to be quite sufficient as a long term solution. Our policy can be found in Appendix A. This is an example of how policy can be an effective

systems administration tool. [3] A little organization can go a long way.

Topology management

Managing TCP/IP network tables can be a sizable task in a very large network. At SAS Institute all network addresses are centrally maintained by the the networking group. Previously, the master table resided on the Apollos in "local.txt" (RFC952) format. From this table were generated the various `/etc/hosts`, **named** and other format tables for all hosts on the network. We found that we were trying to track not just machine names and network addresses with this table, but also other basic information about each host. We needed a better system. System administrators from each different operating system type got together and designed a network topology management system we call **netmain**. **netmain** is a SAS application which maintains hostname, aliases, and IP addresses, mail servers and other standard network information as well as a myriad of other information that administrators want to know about a node on the network. All forms of networking and name service tables are generated directly from **netmain**. In addition to the standard information, **netmain** also tracks information such as the employee to whom the node is assigned and its location, as well as the system and network administrators for that node. For the networking group we record the physical description of a connection, including routers, hubs, and wiring closets. For software support we record operating system level, additional software for which the node is licensed and when the node was installed. Frequently accessed information such as primary user and location are made available though HINFO records in **named**.

Backups

The design of our distributed computing environment meant that every HP 700 computer would have user data on it and would have to be backed up. Our requirements for a good backup system were hefty. This was the one application where it seemed reasonable to purchase a commercial application. There simply was not the time or resources available to implement our own solution from scratch. The backup system had to be able to handle backing up the entire network in an 8 hour period per night, 6 nights a week. It needed a tape management facility that would handle recycling large numbers of tapes. We wanted user initiated restores. We wanted fault tolerant software that would handle losing a tape drive in the middle of a backup cycle. It needed an operator interface for handling errors and tape mount requests. It had to be able to support which ever media we would choose. We wanted to run the same backup system on our other UNIX hosts, as well, and we needed a way to backup all our AFS volumes. We found that there is quite a large number of backup products on the market. Unfortunately, very few of them met all of our criteria. Due to the newness and availability problems of the HP 700 line even fewer vendors expected to have a port done by our September 1991. None of the backup vendors supported backups of AFS volumes. We were very fortunate to find a backup product from Systems Center called **backup.unet** which met most of our criteria and would be available when we needed it. Transarc provided an AFS backup system but it was insufficient for our needs, lacking a tape library management system and backing up only AFS objects. Since **backup.unet** did not handle AFS objects we still had to use part of the Transarc backup system. To backup AFS volumes we use the AFS backup utility to create a compressed image on disk which backup.unet later puts on tape. This system requires lots of extra disk space and makes restores awkward. We do not consider it a very good solution, but have yet to find a better one.

We also needed high density media for our backup system. HP had committed to 4mm DAT tapes as their media of choice. Unfortunately, we did not feel that 1.3 gigabytes/tape would be enough to support an installation of our size. We investigated other options like 8mm tapes but this option required a 3rd party device driver to support it. We found several vendors who were 'working on' such drivers which would be available 'real soon now', but with less than a month before our first 'major' install of 90 workstations we still had not seen an 8mm drive working on our HPs. It seemed a little too early to be able to make a commitment to this media. We had begun hearing rumors that HP was going to be coming out with a compressing DAT drive. Our HP representative said no such drives would be available in the near future. We did find out, however, that these drives could be purchased though 3rd party peripherals vendors. With compression these drives had a capacity of up to 8 gigabytes which was sufficient for our needs. So our media of choice was HP manufactured drives purchased from a 3rd party vendor. We reasoned that eventually we would probably be able to purchase them from HP. We hoped that we would eventually be able to put them under HP maintenance which was very attractive to us due to the critical nature of these drives and HP's excellent onsite service. (It took nearly a year for this to be true!) At the very least these drives seemed likely to work well on HP computers, and although we suffered though problems with early firmware revisions, this seems to have been a good choice.

Problem Management

In any support organization it is important to plan how to handle hardware or software failures. This is especially important in an organization with

relatively few UNIX systems administrators and many other folks for whom UNIX is definitely the "new kid on the block." Our division, the Information Systems Division (ISD), provides centralized computing support services to the entire company and has, over time, developed significant support strategies for traditional computing environments such as MVS, CMS, and VMS. ISD has a Help Desk which fields calls on all supported computing systems, but whose expertise leans most heavily toward mainframes. A relatively high percentage of UNIX calls are routed to UNIX system administrators. We also have a groups of hardware technicians with amazing skill in supporting IBM 3270 terminals, but who had not had much previous experience with UNIX. We needed strategies to fit experience level of support available to us.

Software Support

When a user has a problem or question about an Institute computer system, he typically calls the Help Desk. The Help Desk analyst either addresses the problem himself, or routes the call on to another support person using an internally developed problem tracking system. Some UNIX calls are handled directly by the Help Desk, such as adding accounts and helping users to restore files. Calls not handled by the Help Desk are routed to the person deemed most appropriate by the analyst based on his understanding that particular computer system. It was particularly difficult for UNIX systems administrators since anyone was likely to get a call routed to them on most any subject at any time of day, with the expectation that the call would be handled quickly. If a call was inappropriately routed, a user could be forced to wait an unreasonable amount of time to get his problem fixed. We felt like we were all on Help Desk duty all day every day, which made it very difficult to get anything else done.

To make this system work more smoothly we developed an **oncall** system for UNIX support. In this new system a rotation schedule was established in which one member of the UNIX support staff is on duty or "oncall" at all times during business hours. (The corollary to this is that everyone else is NOT oncall.) We developed a set of support utilities for managing the oncall rotation schedules, substitutions, and a special "oncall" mail alias. When the Help Desk wants to reroute a UNIX call, it is simply routed to "oncall", where the UNIX support person on duty will find and process it appropriately. The oncall person knows he will spend the day handling problems and not much else. This system has greatly simplified the process of solving daily UNIX problems by getting calls quickly to an experienced UNIX administrator who is ready and able to handle them. It allows us to manage the handling of problems without requiring the Help Desk, users and other support staff to keep up with our schedules such as who is out sick or on vacation. It encourages all UNIX support programmers to keep up with what is going on around the network and talk to each other about common problems which crop up.

Hardware Support

Our philosophy for fixing hardware problems is that we should not be doing hardware diagnosis and repair in a user's office. We allotted space in a basement area and purchased 1 spare workstation for every 100 workstations on the network plus a "cloner" machine. The cloner machine is equipped with an external disk cabinet with removable trays which are used to produce replacement system disks. If a user appears to have a hardware failure, a spare component is swapped in by a hardware technician and the suspected failed one is taken to the staging area for thorough diagnosis by an HP service engineer. All of our HP 700s are configured with 2 400 megabyte internal disk drives. Data is distributed across these disks such that all files unique to a particular user's machine are contained on the second drive. This includes not only the user's home directory but also their system mailbox, crontabs, backup configuration files, etc. The system disk can then be recreated from a standard HP-UX distribution customized by our automatic installation tools. Only if a user loses his extra disk drive does a hardware replacement take more that a few minutes. To make this process less painful the backups tables necessary to restore the entire local disk are copied to the system disk each evening. In the case of a lost local disk the technician must install a blank second disk, recreate the backup tables from the system disk copy, and restore user data from tape. This system has significantly reduced user downtime due to hardware failure.

We also set up software tools which increase the independence of our hardware technicians. We created a special technician account whose login shell presents them with a menu of common tasks related to hardware support. When a technician logs in he sees:

```
******************
* Tech Login Menu *
******************

 1) Quit
 2) Reboot
 3) Shutdown
 4) Update Inventory/User Info
 5) Ping Another System
 6) Show Network Info
 7) Change IP Address and Hostname
 8) Exit to Shell, ksh
 9) Clone Disks
10) Reporting

  Select Action by Number:
```

These tools improve the consistency of support and reduce the need for intervention by systems administrators.

User Account Management

One of the requirements of the distributed computing design for our development network was a unified users account structure in which users appeared to have a single account with the same password and home directory throughout the network. We wanted to avoid reliance on a central network resource for basic login abilities. On the Apollo network, if the account registry server was unavailable or confused (as it often was), no one on the network could login. This project, like backups, was one for which we strongly hoped to find a commercial solution which would help us to manage user accounts on our other UNIX platforms. HP had announced a product called "Password Etc", but this seemed to bear too much resemblance to the Apollo registry to be interesting, and was not available in time for us to meet our deadlines. It also did not help us out with our heterogeneous UNIX problem. We did not seriously consider solutions such as NIS due to known problems with this model operating on such a large scale.

We found a terrific solution in a package called ACMAINT developed at Purdue University. [4] ACMAINT maintains user account related files, such as `/etc/passwd` and `/etc/group`, out of a central data base via a server which runs on each machine. ACMAINT provides a mechanism for managing account creation, deletion, moving home directories, etc. via simple utilities. Only programs which actually modify the password and group files needed to be replaced on each machine. All account transactions are queued so that if any machine is down at the time of an update, the update will be resent later. ACMAINT operates transparently to a user and results in normal standalone password and group files. It required only a moderate porting job to get it to run on HP-UX. Modifications to the program included the addition of "hostclass" support, as well as management of AFS accounts. We also made changes to make user updates such as a password change happen immediately on the local machine instead of waiting for an update from the database daemon. (Very long update times for password changes caused by busy ACMAINT database servers occasionally confused our users.)

Console Management

Console management as a project was not something that occurred to us in the early stages of this conversion project, but turned out to be significant. Floor space in our data center was at a premium. We were already gearing up to go vertical by storing our initial 30 HP servers in 3 layers on industrial strength steel shelves. One day someone came to us and asked where were we planning to put the 30 ASCII terminals to be used as consoles for our servers? "Consoles! We hadn't thought about that." We turned to the LISA proceedings in hope of a solution. Fortunately, as with user account management, this problem had already been solved. Tom Fine at Ohio State University [5] developed a console server package which allows one to manage consoles by connecting the console ports of several machines into the serial ports of another and using his **conserver** software to manage access to them. **conserver** also logs all activity on each port. The client program, **console**, allows a systems administrator to connect to the console of a server over the network. Although written on a Sun it was a fairly simple matter to port this application to HP-UX. Since HP did not (and still does not) provide serial multiplexers for the 700s beyond the two ports which come with each machine, we had to scrounge an old HP300 for this purpose. In our data center operator console area we have an HP700 graphics workstation on which the operators monitor our HP network servers. On this machine they have one window for each server connected to the console server process for that machine. This solution not only makes efficient use of data center space, but also of a system administrator's time by allowing him to connect directly to the console of a machine without leaving his office.

Printing

On the Apollo network users tended to have lots and lots of small printers scattered everywhere. Everyone, of course, wanted his favorite neighborhood printer available from his new HP. Printing under HP-UX is a hybrid of System V and BSD print facilities. The local users print utilities are the System V **lp** suite. These utilities are enhanced by adding the BSD lpd protocols to create **rlp** which sends line printer requests to a remote printer. This is somewhat better than **lp**, but still requires that the user know what printers with which options are available in some remote machine. It is not even as functional as a standard BSD printing system. We came up with an alternate approach to network printing. Suppose that instead of having all the printer description files local to each machine one simply had a print utility that would query a centralized printer daemon which would provide all the information about printers available throughout the network. With this approach one would only have to change the configurations of the central printer server daemon to make new print resources available to the network. This model followed the guidelines of not having to "touch" each workstation in order to effect network-wide changes. We called this system **nlp**. To the user, **nlp** looks like the System V **lp** command, with individual command line options defined to take advantage of different capabilities of

our various printers. Since options can very with different printers, **nlp** provides a query option which describes what features are available for each printer. The **nlp** system uses the BSD **lpd** protocol to send jobs to the actual print spooling system. This means that printers can be attached to any machine on the network which supports the BSD remote printing. In our environment this includes not only UNIX, but other systems such as VMS and VM. To add a new printer in the network, a systems administrator has only to set up the print spooler on the machine to which the printer is attached and update the **nlp** network print daemon's configurations. **nlp** provides a highly distributed network printing system which is very easy to support even on a scale of 1000 workstations. **nlp** is described in detail in a separate paper. [6]

Super User Access

Controlling access to super user privileges is a real challenge in any large UNIX network. Knowledge of the root password provides unlimited privileges and is impossible to contain without changing frequently. Vendors typically require that you run many system utilities as root, but do not provide any utilities for super user access. On the Apollo network we used a locally developed utility, **ru**, to limit root access. **ru** was based on the **sudo** program described by Evi Nemeth in "The UNIX System Administration Handbook." [7] We ported **ru** to the HP network and added hostclass support. The specific privileges granted to each user are contained in a configuration file which lists which commands a user may execute on a specific machine or class of machines. Alternatively, one may list which commands a user may not execute (for example everything except shells), or a '*' indicating unrestricted access. All commands executed by **ru** are logged.

Installation Logistics

The December installation of 300 machines was a triumph of logistical planning. In addition to installing the new workstations the old Apollos had to be de-installed. A new building was completed on campus about this time and Institute management decided that, as long as we had the network down, it would be a good time to move the 88 people in the Quality Assurance department to a new building. This, of course, meant that all 5 floors of the existing software development building had to be rearranged to fill the newly available offices, adding 120 more office moves to the project. The decision to add office moves was made after the location/hostname/address information was send to HP's Integration Center so that more that half the new computers had to be reidentified and retagged. Since our TCP/IP addresses are geographical all these machines had to have their network addresses reconfigured as well. The networking group had to re-wire most of the building. Changes to the exact locations were still being made withing hours of the beginning of the installation.

Different phases of the installation and moves were handled by specialized teams made up of 130 employee volunteers. Each type of team was given special training and instructions. Existing equipment was tagged with large red and green stickers to indicate whether it was moving or being removed. Teams operated around the clock from Wednesday night until Sunday afternoon. First, de-installation teams went around and formatted disks and de-installed hardware that was scheduled to be removed. For hardware that was being moved, teams had to shutdown and disassemble machines, putting the smaller parts in large Ziplock bags. After the de-installation teams finished, professional movers managed by Institute Facilities crews moved furniture and computers to new locations. Next, re-installation teams came along to reconnect all the computers.

Interleaved with the move process was the installation of new HP workstations. The building was divided up into geographical areas with a Floor Captain assigned to each area. Floor Captains had the complete information on all installations in their area, and coordinated several installation teams who unboxed and connected each new workstation. Each Floor Captain was assigned one or two software reconfigurers who changed network addresses and ran installation verification scripts. Reconfigurers were required to have some previous experience with UNIX. After the installation teams finished, trash teams came around and removed boxes and packing materials. Each Floor Captain was issued a walkie-talkie to communicate with a central Dispatcher who logged all problems and requests (e.g., power strips, extra screw drivers). UNIX systems programmers and network specialists congregated in a conference room near the Dispatcher waiting to be sent off on a problem requiring their particular expertise. All activities and teams were meticulously planned and schedules followed precisely. And, incredibly enough, it worked. On Monday morning 300 programmers returned to work to find that a new HP workstation had miraculously appeared where their Apollo had been.

Conclusion

Purchasing the very latest cutting edge technology may be desirable from a price/performance standpoint but can present difficult support problems. The downside of purchasing new technology is that you may be on your own for any additional programs that you need. Thrid party solutions are likely to be scarce. Time and again when contacting vendors we heard comments like, "We'd like to port our software if only we could get our hands on a

machine'' or ''It's a little early to tell. We might have something next year.'' In order to get a version of the backup software in time to meet our deadlines, we loaned a one of our seed development machines to our backup vendor for a month. We also had to ''encourage'' HP and Transarc to get a version of AFS out for the 700s.

Another feature of buying 'bleeding edge' technology is that you will get to help the vendor find all those little problems that only appear at customer sites. When the HP 750 servers were still very new the differential SCSI controller would randomly trash our disks. HP could not repeat the problem in their labs. They finally sent an engineer down to us with his SCSI analyzer to look at our machine and ended up taking our server back with him for further analysis. Probably the most memorable incident involved FDDI controller cards. After we had our servers for a while we received the first shipment of FDDI controller cards. When the HP service engineer went to install these cards he discovered that they would not fit in the server's card cage. It seems that there was this capacitor that stuck out just a little too far.... The boards were quietly whisked back to HP for reengineering. Apparently even labs inside HP were having trouble acquiring production servers.

In developing our systems administration utilities we learned some very important lessons. First, replicate, replicate, replicate. While centralized network services are easiest to support, they can represent single points of failure which are not fault tolerant enough for large networks. Eventually almost all of major client/server applications had replicated servers added to them. If you do not own a full set of LISA proceedings, go and buy them. They are terrific sources of solutions and good ideas. Take the time to do it right. This may seem obvious, but so often it seems that we develop solutions which are just sufficient to solve the current crisis. The time that we spent working on projects for our HPs was time we did not spend supporting our existing network. Service requests were late, problems were solved slowly, users complained, morale fluctuated. The end result, however, is a highly supportable network which runs smoothly (for the most part). It was worth all the time that we put into it.

Acknowledgements

Heartfelt thanks to the UNIX Support group, Mike Beach, Mark Fletcher, Ken Howell, Mike Mitchell, Jeff Phillips, Rick Tatem, and Mike Shaddock and to everyone else who made this conversion project a success.

Author Information

Helen E. Harrison is the UNIX Support Manager at SAS Institute Inc, where her group supports a network of over 800 UNIX workstations, and servers. She has been involved in UNIX systems administration for 10 years and holds a B.S. in Computer Science from Duke University. Reach Helen at SAS Institute Inc, SAS Campus Drive, Cary, NC 27513; or by e-mail at heh@unx.sas.com.

References

1. Mark Fletcher, ''doit: A Network Software Management Tool,'' *LISA VI Proceedings* ., Long Beach, October, 1992.
2. Helen E. Harrison, Stephen P. Schaefer, and Terry S. Yoo, ''Rtools: Tools for Software Management in a Distributed Computing Environment,'' *Proceedings of the Summer USENIX Conference*, pp. 85–93, San Francisco, June, 1988.
3. Elizabeth D. Zwicky, Steve Simmons, and Ron Dalton, ''Policy as a System Administration Tool,'' *LISA IV Proceedings*, pp. 115–123, Colorado Springs, October, 1990.
4. David A. Curry, Samuel D. Kimery, Kent C. De La Croix, and Jeffrey R. Schwab, ''ACMAINT: An Account Creation and Maintenance System for Distributed UNIX Systems,'' *LISA IV Proceedings*, pp. 1–9, Colorado Springs, October, 1990.
5. Thomas A. Fine and Steven M. Romig, ''A Console Server,'' *LISA IV Proceedings*, pp. 97–100, Colorado Springs, October, 1990.
6. Mark Fletcher, ''nlp: A Network Printing Tool,'' *LISA VI Proceedings* ., Long Beach, October, 1992.
7. Evi Nemeth, Garth Snyder, and Scott Seebass, *UNIX System Administration Handbook*, Prentice Hall, 1989.

APPENDIX: GUIDELINES FOR PROGRAMS IN /USR/LOCAL

The following is a set of guide lines for the management of local programs, administrative tools, and libraries (all of these shall be referred to as "tools").

- /usr/local/{bin,etc,adm,include,lib,man,src,beta} (see below for a description of each subdirectory) shall be the official repository for local "supported" tools. A tool can be put in /usr/local if it meets the following requirements:
  - There must be a manual page for all of the relevant parts of the tool. For example, emacs has at least two auxiliary programs, loadst and etags. There should be manual pages for emacs, loadst, and etags. Emacs also has many LISP libraries, which in this case would not need manual pages.
  - There must be a source directory for each tool. In that directory there must be a Makefile with the following minimum targets: all, install, clean. A system administrator should be able to go to the tool's source directory, type "make clean" and "make install" and have everything work appropriately (all unnecessary files removed by "make clean", all necessary files installed in correct places by "make install"). The Makefile must also be parameterized for easy future customization. An example of this would be a macro called BINDIR which points to the appropriate binary directory (typically /usr/local/bin).
  - There must be someone willing to support the tool. In this instance support can mean various things, where the minimum amount of support is taking HelpDesk calls on the program and the maximum amount is actually fixing bugs in the tool. The support person is also responsible for getting updates to the tool as deemed necessary.
  - There must be a README.SAS file in the top level source directory for the tool. The name README.SAS was chosen over README since many of these tools are public domain tools that already have README files. This README.SAS file should have a line that says Supported-By: in it. This will be referenced by the HelpDesk in routing calls in this tool. If the source to this tool does not actually reside in /usr/local/src, the real source location should be listed. The README.SAS file should also have information about what local changes may have been made to make the tool work in the SAS environment, where it came from, special installation and configuration instructions, etc. It should also have a list of all the tools that are generated into either of /usr/local/etc or /usr/local/bin.
- /usr/local/contrib/{bin,etc,adm,include,lib,man,src} shall be the official repository for local "unsupported" tools. A tool can be put in /usr/local/contrib if people want it to be publicly available. No HelpDesk calls will be accepted on tools in /usr/local/contrib, nor is any system administrator guaranteed to work on any of these tools. In addition, these tools may be backed up infrequently. If a tool is so important that it must always be there, then it is important enough for someone to volunteer to support it, and it can be moved to /usr/local.

Description of /usr/local/subdirectories:

/usr/local/bin	local generally used executables
/usr/local/etc	local administrative tools
/usr/local/adm	other local administrative tools and log files
/usr/local/include	local include files
/usr/local/lib	local libraries and support programs
/usr/local/man	local manual pages
/usr/local/src	source for all of the above
/usr/local/beta	place to beta test tools

Mkserv – Workstation Customization and Privatization

Mark Rosenstein & Ezra Peisach – MIT Information Systems

ABSTRACT

Adding any form of service, such as NFS, dial-in lines, or time synchronization, to a workstation or server is an operation which requires a knowledge of the operating system and the service in question. As more platforms are introduced into an environment the knowledge base and training required also increases. AT MIT/IS we have developed a tool called *mkserv* which handles most of the service customization issues on a variety of platforms running several versions of UNIX in the Athena Computing Environment. This process is tied into the software release process and allows us to automatically update over 90% of our 1200 workstations on campus while preserving and updating the owners' customizations. For the private workstation owner, we use *mkserv* to add special configurations to their machines, such as serial dial-in capability, which is then preserved through software updates.

Introduction

All software on Athena workstations at MIT is tightly controlled by the centralized file service. We use a cookie-cutter approach to make workstations identical except for one configuration file which defines the hostname, IP address, and a couple other parameters. Our 150 central servers are essentially the same as workstations with local modifications to the system software or hardware for whatever service it is supporting.

Mkserv (pronounced make-serve) was developed to satisfy several needs in the Athena environment. We needed a uniform method for the configuration of servers such that all servers of a given type are identical and have nothing missing. In recent years, more and more machines have been placed in private areas with private owners who want their machines to provide services that we have not thought of. We needed a way for private owners to define both private services and customizations that are preserved across software updates. Extending *mkserv* to do this as well was the logical approach. This has allowed us to add such services as serial line dialup in the hands of the private (non-technical) owner where they can learn what to do by a simple phone call, and are unlikely to make a mistake. This saves much time and effort on our operational and support staff.

Mkserv is a program which stores state about a machine's configuration and can reconfigure a machine properly through a robust scripting language. Features include inter-service dependencies, automated configuration file changes, service addition and removal scripts, hardware platform specificity, and user extensibility.

We use an automated software update strategy in which *mkserv* plays a key role. Due to this fact, most private workstation owners are willing to use this auto-update strategy as well. The end users are happy because we no longer break their configuration. We are happy because we no longer have to spend time scheduling for visits to the customers for updates.

Environment

MIT has over forty ethernet subnets linked by an FDDI fiber-optic backbone. While there are several major computation centers and numerous labs and departments with their own computers, most academic computing at MIT is done on Athena workstations.

Athena has about 1200 machines including DECstations running Ultrix 4.2a, RISC/6000's running AIX 3.1, VAXen running 4.3 BSD, and IBM PC/RT's running AOS 4.3. This is expected to expand to other manufacturers and varieties of Unix in the future. About half of these workstations are in unstaffed public areas accessible 24 hours a day. The remaining machines are in private offices of faculty, graduate students, and our staff. These workstations are supported by 150 servers, largely running on the same hardware as the workstations. Because of hardware pricing at the time most systems were purchased, our workstations have small local disks and most software is used directly from fileservers.

A number of network services are in use to meld a thousand workstations into a coherent distributed system. File service is primarily AFS 3.1, although we are still making some use of NFS servers and even some RVD[1]. There are numerous replicated copies of the system software delivered on read-only filesystems. We refer to these filesystems as system packs. Workstations are configured to use

copies which are topologically close to them on the network. We are using Kerberos [2] version 4 for authentication of most of our network services. Hesiod [3] layered on top of the Domain Name Service provides name service. Zephyr [4] provides real-time notification. Time is synchronized by *timed* within each subnet and NTP between subnets. The various services are configured and coordinated by Moira[5]. All of this is run by a system support staff of six for workstations and operations support staff of six more for servers.

Athena Software Releases

An Athena software release comprises both the basic operating system and all Athena additions. For the BSD-based platforms, we compile everything from scratch. On the other platforms, we start with the vendor-supplied operating system and layer our changes and additions on top of it. A few local changes require replacing vendor binaries instead of just providing new programs under */usr/athena*. [6]

Once or twice a year we update all of the workstations on campus to a new campus-wide release. Because this affects a thousand workstations, it must be completely automated. In the days when hand-work was required to update workstations, it often took several months to get everything updated. This made it impossible to put out a release between semesters and have all public workstations and most private ones running the new software before the start of the following semester.

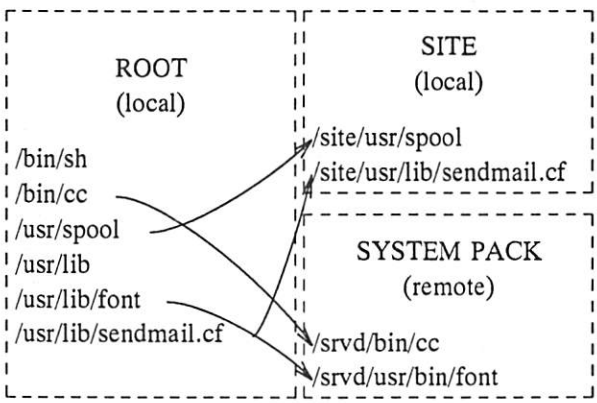


Figure 1

Workstations now update themselves automatically when they are not in use. This update takes only a few minutes after which the workstation will reboot and be back in service. An idle workstation periodically compares the version of software on its local disk with that on the fileserver. If the two differ, a script is run to update the workstation. This script copies a minimal set of binaries to the local disk, and creates symbolic links (sym-links) for the rest to directories mounted from a fileserver. The choice of whether a program is copied local or not is the same for all workstations, made when the release is built. See Figure 1 for a sample of some file locations and sym-links between the local disk partitions **root** and **site** and the remote **system pack** mounted on */srvd*. Our release engineering group decides this based on what is needed at boot time to bring the machine up on the net, what is needed to diagnose hardware and network problems, and a few things brought local for performance reasons. The decision to auto-update is actually a configuration variable, but very few private workstation owners (less than 10%) do not trust the automated procedures. Central servers are configured not to auto-update because that would result in an unscheduled service outage of several minutes.

Automatic installation and updates are easy for cookie-cutter identical workstations, but machines which provide a service have many files stored locally and may have configuration files in system directories that the update process would overwrite. This is where *mkserv* comes in. Besides initially configuring a service, it knows how to recover all of the state and reinstall a service following an update. In addition to setting up configuration files, *mkserv* will copy to the local disk any binaries needed by the service which are normally left on the fileserver.

Customizable Services

We actually lump together several different things that we call "services" in this context. First there are the actual services. For example, NFS file service which requires copying to the local disk *rpc.mountd, rpc.quotad, nfsd*, and a couple of other binaries, seeing that the proper daemons get started in the proper order at boot time, and initializing an *exports* file. Besides file service, other services not usually enabled on workstations are local mail delivery, discuss [7] (bulletin board), printing through directly connected printers, Kerberos server, and others.

Then there are other configuration changes that we loosely call services. A common one is remote access. By default, workstations are configured not to allow any kind of incoming remote access: not telnet, rlogin, nor even ftp. Enabling the service **remote** will copy the necessary daemons to the local disk, then change */etc/ttys* and the configuration of the inet daemon. It will also remind the user to change the root password, since all public workstations have the same (published) root password, which is fine if the only access to the workstation is through the console, but a problem if multiple people can login at the same time. Another similar service is enabling serial lines for dialin and dialout. Something a little different is increasing system resources. We have a service which allocates additional swap space on the local disk and brings in a kernel with larger tables for people who run large software packages requiring more resources than configured into the standard workstation.

Finally, some workstation owners have one-of-a-kind changes that they want on their workstation and they want it to continue working across system updates. It is possible to define a service which *mkserv* will manage. If done according to the published guidelines, there is a good chance that new system releases will not break such services. An example of this might be a workstation which runs a trick finger daemon to return additional or false information.

Requirements

For the most recent rewrite of *mkserv* (we are on version 3), we came up with the following requirements.

Updates must be as automatic as possible. No hand-work may be required. It must be possible for *mkserv* to re-install services whenever a workstation updates. If an error is encountered, it should make a best effort to leave the workstation in working condition. Also the operator should not be required to know anything about the service. If any additional information is required, *mkserv* must ask for it in a sensible way and sanity check the answers it is given.

When a workstation updates, all services must be preserved. Any combination of services, whether centrally defined or custom, should continue to work on future releases. Any configuration files that must be kept across updates must be stored in a location that will not get clobbered.

Workstation owners should be able to easily specify their own customizations. Customization definitions may be kept on the local disk (for convenience) or a fileserver (for safety). *Mkserv* must remember where the definition of each service is kept, and verify that it can find all services before it starts.

After a service is installed, the workstation should have no dependencies on other external services. All necessary files must be copied to the local disk. It is important that there not be inter-dependencies between services so that there is no deadlock starting up following a campus wide power hit. Also, an idle workstation or server should not present any load on other services.

It must be possible to tell what services are installed on a machine. This includes identifying what version of a service is installed, since we may patch them over time and not all services may be have been installed at the same time.

Finally, *mkserv* must make a reasonable effort to determine if it can run to completion before it starts. We're not trying to solve the halting problem here, just making sure than any needed network file systems are accessible and other resources such as disk space are sufficient.

Solution: Mkserv

Mkserv is as much a methodology as an implementation. The code is pretty simple. It was originally a shell script, but now is largely in C code. However it still uses scripts for service specific parts of the process. Basically it consists of four parts:

1. Checking dependencies on other services
2. Verifying that all needed resources are available
3. Service specific configuration
4. Copying the correct files to the local disk

With the exception of step 2, all of this has been present since the version 2 of the software. However, the system has gotten more intelligent and featureful over time.

Mkserv has only a simple command line interface. Since it is only invoked by the end user to add or remove services (i.e. very seldom) there is just not enough call for a fancy user interface. Most invocations will be automatic from the larger update process which installs a newer system release on the workstation.

The first version of *mkserv* (written in Bourne shell script) made some very implicit assumptions about what combinations of services were allowed and how the disks were laid out. Specifying some combinations of services worked and others did not. If the disk layout was changed (for instance, */usr/spool* is a sym-link to a directory on another partition instead of residing on the */site* partition) then it was likely to fail. These problems are fixed in the current version.

Services are very easy to define. A service definition consists of four files: dependency, sync, add, and delete. These files are all found in a common directory for centrally defined and maintained services. For instance, service **nfs** is defined in the files *nfs.add, nfs.del, nfs.dep,* and *nfs.sync*. If a user defines his own service, he may specify which directory *mkserv* should look in to find the service definition files. *Mkserv* will remember which of these directories are on network filesystems so that at future invocations it can verify that all needed filesystems are accessible when it starts.

Dependency checking is very straight-forward. A configuration file for each service lists which other services it depends upon. If you attempt to install a service without a necessary prerequisite service, that other service will automatically be included. So when our **eos** [8] server (Networked Educational On-line System) which uses Sun RPC is installed on a machine, it depends on service **rpc** just as service **nfs** does to get the portmapper and related RPC files.

For now, checking for necessary resources is limited to a check that all needed remote filesystems are accessible. *Mkserv* checks that it can find all of it's configuration and service definition files, but does not check that all binaries that need to be

copied to the local disk are present. While it is possible to check disk space as well, that calculation is complicated because often several services want the same files copied local, and often *mkserv* is run for an update when services are already installed. We have not found the lack of this check to be a problem in practice.

Mkserv now uses a utility called *synctree* to handle the data moving. Earlier versions used track [9] a similar but less flexible tool written at Athena. *Synctree* was written at the MIT Laboratory for Computer Science. It reads a configuration file specifying what files or portions of a destination filesystem hierarchy should be copied from or have sym-links pointing to a source hierarchy. This tool is versatile in how it can be instructed to follow or overwrite sym-links on the source or destination. We use it for such diverse tasks as copying a source tree, filling a build tree with sym-links to a source tree, and installing a machine from scratch.

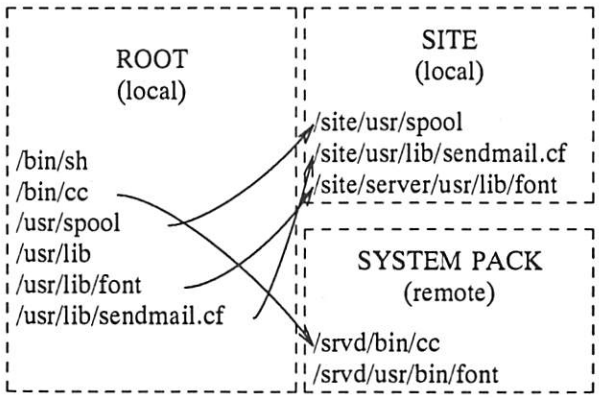


Figure 2

Synctree is used in a two-step process. In the first step, any necessary files are copied from the remote filesystem to a local disk, generally the */site* or */var* partition. The *synctree* configuration for this step is generated by concatenating a generic header with service-specific lines for each service supported, then running that through the C pre-processor with the name of each service defined. This effectively generates a list of every file and directory to be copied local. *Synctree* is then run to do this copy. In the second step, a static configuration is used as input to *synctree* to instruct it to find where on the root filesystem the files copied above are usually found, and put in an appropriate sym-link.

For example, on most workstations */usr/lib/font* is a sym-link on the root partition pointing to a directory on a remote filesystem. When creating a print server, in the first *synctree* pass all of the fonts will be copied to the local */site* partition as */site/server/usr/lib/font/**. In the second *synctree* pass the */usr/lib/font* sym-link will be changed to point to the local copy instead of the remote copy. Compare Figure 1 with Figure 2 which shows the changes after running "mkserv print". By using this two-step process, usage on the root partition does not change when *mkserv* is run, and all binaries are still available whether on the local disk or remote system pack.

The service-specific configuration is still done via Bourne shell scripts. Each service must have an add script which is run when that service is added and again each time it is updated. It may also have a delete script which is run when the user wants to remove support for that service from their workstation. When running each script, a number of environment variables are set to provide useful information. These include the path to the system software on the fileserver, the root of the local copy of the system software, the root of the partition containing the system software, the name of the output log file, the release version number of the system software, the name of the workstation architecture (i.e. VAX, RT, SUN4, etc), a directory the service may stash configuration files in, and a file in which to put configuration change commands. Some of these values change from one operating system to another, so by using them rather than hard-coded paths, the scripts are portable.

One of the scripts' outputs is the configuration change file which contains commands to change common parts of the workstation configuration. Following execution of each service add script, *mkserv* interprets this command file to finish the configuration changes. The commands are:

adduser: adds an entry to the password file

deleteuser: removes an entry from the password file

addservice: adds an entry to the *inetd* configuration

deleteservice: removes an entry from the *inetd* configuration

switchservice: makes an *inetd* configuration entry switched

conf: sets a variable in our *rc.conf* file

By having *mkserv* accept these commands from the service-specific scripts, it frees up the writer of the service scripts not to have to figure out how to do these common tasks, each of which requires locking and/or consideration for possible interactions with other services or operating system specifics.

Example: NFS

Here is a complete example for NFS. Note that we run a modified NFS [2] at Athena which uses Kerberos authentication at mount time and UID mapping between the client and server. First, the dependency file indicating that **rpc** is required:

```
/* $Revision: 1.1 $ */
#if defined(nfs) && !defined(rpc)
rpc
#endif
```

Figure 3 shows the add file run when the service is added or updated. This sets the proper configuration variables so that */etc/rc* will start the daemons and verifies that authentication and exports are setup correctly. Note the line with **$Platforms** which indicates on what kind of machine this may be used. The del script run when the service is deleted is much simpler:

```
#!/bin/sh
# $Revision: 1.4 $
# $Platforms: decmips,vax,rt $
echo "conf NFSSRV default" >> ${CONFCNG}
echo "conf RPC default" >> ${CONFCNG}
exit 0
```

And finally, Figure 4 shows the sync script which tells *synctree* what to do. Note that it is conditionalized for which operating system is being run. The check for **ops** near the end is a pseudo-service we put on all of our operational servers that includes tools and monitoring programs.

Other Software

There are other packages available that implement parts of this functionality, but none that do all of it, and certainly none that fit into the Athena environment without needing some customization.

Package [10] from Carnegie Mellon University relies on too much environment to run (file service, etc). The configuration files are too centrally managed and is not flexible enough for end users to easily use. Configuration files must be hand edited by too often naive users with devastating consequences.

Depot [11] was also originally from Carnegie Mellon, now marketed by Transarc. Depot currently assumes software will be stored under a */usr/local* hierarchy. Depot is useful for third party applications where one is mainly interested in copying binaries locally for performance or making available applications via sym-links. This application deals with version numbers between applications. Future work will probably allow for configuration of other software on the machine.

```
#!/bin/sh
#
# $Id: nfs.add,v 1.3 92/05/09 epeisach Exp $
# $Revision: 1.3 $
# $Platforms: decmips,vax,rt $
if [ ! -d /${SITE}/usr/etc ]; then
    mkdir /${SITE}/usr/etc
fi

echo "conf NFSSRV 8" >> ${CONFCNG}
echo "conf RPC true" >> ${CONFCNG}

(klist -file /etc/athena/srvtab -srvtab | grep '^rvdsrv ') >/dev/null 2>&1
if [ $? != 0 ]; then
    echo "===> Remember to get a srvtab with a 'rvdsrv.${HOST}' key" >> $LOGFILE
fi
if [ ! -s /etc/exports ]; then
    echo "===> Remember to set up /etc/exports" >> $LOGFILE
fi
if [ ! -s /usr/etc/credentials ]; then
    echo "===> Remember to set up a credentials \
      file in /${SITE}/usr/etc" >> $LOGFILE
fi

cat >> ${LOGFILE} <<EOF
===> Remember to arrange the appropriate Moira
===> entries for NFS service.  Please make sure
===> NFSSRV is set to an appropriate number in
===> /etc/athena/rc.conf.
EOF
exit 0
```

Figure 3

Lessons Learned

We've learned a lot in five years of trying to manage a large number of workstations with a very small staff. *Mkserv* is our solution to automating the handling of a number of special cases so that they no longer require any special handling.

One of the most important things we have found is the truth of the KISS (Keep It Simple, Stupid) principal. We have not tried to make one monolithic program that will do everything. Instead we have made simpler, flexible tools that can be combined to solve the task at hand (for instance, using *synctree* and shell scripts within *mkserv*). Keeping the system as simple as possible makes it less likely that there are design flaws or bugs. It makes it easier for the developers and maintainers to understand so that it does not take a wizard to do anything with it. Changes can be integrated cleanly rather than growing as a unmanageable tangle on the side of the code. Simple systems are easier for users to understand so they can figure out what a program does and how to use it.

Everyone makes mistakes. Not just the users, but consultants, operators, even your network wizards. By automating common operations you make it less likely that these will be done wrong. It avoids people having to remember or look up details about configurations. It also assures that it is done the same way every time so that you can judge the impact of future changes without worrying about how someone might have configured the service.

End users tend to be non-technical. Requiring them to hand-edit critical files is a formula for disaster. Certainly, there are clueful users and most can follow directions, but there are enough who cannot to be a major support problem. Mis-formatting some of these configuration files can result in a workstation that must be re-installed from scratch.

If you do not make provisions for the sophisticated workstation owner (hacker) in workstation customization, then they will make changes which cannot be supported in the future. This will ultimately cause them to be very unhappy with new releases because the update will break their environment. If

```
/* $Id: nfs.sync,v 1.1 92/05/09 epeisach Exp $ */
/* rpc should be it's own service maybe? */
copy usr/athena -f
copy usr/athena/etc -f
copy usr/athena/etc/mkcred -p -f
copy usr/athena/etc/mountd -p -f
copy usr/athena/etc/nfsc -p -f
copy usr/athena/etc/rpc.mountd -p -f
copy usr/athena/etc/rpc.rcquotad -p -f
copy usr/etc/rpc.rquotad -p -f
/* Don't force new credentials file into place */
copy usr/etc -f
copy usr/etc/credentials* -p
copy etc -f
copy etc/athena -p
copy etc/athena/shutdown_notify -p -f
copy etc/athena/zshutdown_notify -p -f
#ifdef ultrix
copy usr/etc/nfsd -p -f
copy usr/etc/lockd -p -f
copy usr/etc/statd -p -f
#else
copy etc/nfsd -p -f
#endif
#ifdef ops
copy usr/etc/nfsstat -p -f
copy usr/etc/rpcinfo -p -f
copy usr/etc/showmount -p -f
#endif /* ops */
```

Figure 4

users know their customizations will be preserved, then they are more likely to trust automatic updates. In the early days of the Athena workstation environment, most private workstation owners were reluctant to take new releases because they had made various local changes to the configuration which would be overwritten by the update. Our current update procedures and *mkserv* allow customizations that will be preserved across updates. When all of the workstations in the field stay with the current release, it makes our support job a lot easier.

So *mkserv* is an important part of our strategy for installing, updating and maintaining our workstations in a way that keeps our customers happy and is easily supportable.

Acknowledgments

Mkserv has had several authors over the years. *Mkserv* was originally written by Mike Shanzer in January 1989 and was rewritten by Ezra Peisach and Richard Basch in Fall of 1989 to use *Synctree* and a shadowing approach. Further enhancements were continued by Richard Basch adding in new features such as service dependencies, deletions and local customizations. It was rewritten this year by Ezra Peisach to allow for external configuration files written by end users and to be more robust in our multivendor environment.

Many thanks to the Operations staff, for which *mkserv* was originally written, in being guinea pigs during the development of configurations for them.

Thanks to Tim Sheppard and Stan Zanarotti of the MIT Laboratory for Computer Science for allowing us to modify an internal program to create *synctree*.

Availability

A key component, *synctree* is not currently redistributable. We hope to re-write it soon and remove the current source restrictions. If and when that happens, we will announce it to the net. If you still want *mkserv*, we can give it to you. Contact info-athena@athena.mit.edu for information.

References

1. M. Greenwald, *Remote Virtual Disk Protocol,* M.I.T. Laboratory for Computer Science, 1985.
2. J. G. Steiner, B. C. Neuman, and J. I. Schiller, "Kerberos: An Authentication Service for Open Network Systems," in *Usenix Conference Proceedings*, M.I.T. Project Athena, Winter, 1988.
3. S. P. Dyer, "Hesiod," in *Usenix Conference Proceedings*, M.I.T. Project Athena, Winter, 1988.
4. C. A. DellaFera, M. W. Eichin, R. S. French, D. C. Jedlinsky, J. T. Kohl, and W. E. Sommerfeld, "The Zephyr Notification System," in *Usenix Conference Proceedings*, M.I.T. Project Athena, Winter, 1988.
5. M. A. Rosenstein, D. E. Geer, and P. J. Levine, "The Athena Service Management System," in *Usenix Conference Proceedings*, M.I.T. Project Athena, Winter, 1988.
6. G. W. Treese, "Berkeley Unix on 1000 Workstations: Athena Changes to 4.3BSD," in *Usenix Conference Proceedings*, M.I.T. Project Athena, Winter, 1988.
7. K. Raeburn, J. Rochlis, W. Sommerfeld, and S. Zanarotti, "Discuss: An Electronic Conferenceing System for a Distributed Computing Environment," in *Usenix Conference Proceedings*, M.I.T. Project Athena, Winter, 1989.
8. W. Cattey, "The Evolution of *turnin*: A Classroom Oriented File Exchange Service," in *Usenix Conference Proceedings*, M.I.T. Project Athena, Summer, 1990.
9. D. Davis, *Project Athena's Release Engineering Tricks*, M.I.T. Project Athena, 1989.
10. R. Yount, *Package (Obsolete internal documentation),* Academic Services. Carnegie Mellon University, 1985.
11. W. Colyer and W. Wong, "Depot: A Tool for Managing Software Environments," in *Usenix LISA Conference Proceedings*, Computing & Communications. Carnegie Mellon University, Fall, 1992.

Author Information

Mark Rosenstein is a senior member of the systems development group at MIT Information Systems, formerly Project Athena. He is project leader for Moira, Athena's configuration management system, as well as being involved with Athena's custom login, Release Engineering, software installation, network booting, and too many other efforts. Mark is also responsible for the slide-rule mode on xcalc. Mark may be reached at mar@mit.edu.

Ezra Peisach is now pursuing a master's degree in Biophysics. Between his Bachelor's in Chemistry and his graduate studies, he dabbled in programming with the System Development group at MIT Project Athena where he has been involved in a variety of projects related to operating system support including Release Engineering, porting Athena to DECstations, and co-supporting for the X consortium the R3 apa16 X server. Ezra can currently be reached at epeisach@mit.edu.

AUTOLOAD: The Network Management System

Dieter Pukatzki – Leading Edge Technology Transfer
Johann Schumann – Institut für Informatik

ABSTRACT

A variety of network-based software loading tools have evolved over the past few years. They are mostly vendor-specific and represent extensions of removable media loading tools. The implementations usually rest on shell level r-commands as well as on Simple Network Management Protocols (SNMP), Some of them use the public domain program *rdist*. Virtually all are partial solutions for this very complex problem. After having implemented a high level "one vendor" system, the authors decided to develop another one on a low level, capable of serving the entire UNIX community. This paper focuses on the features and problems one needs to understand when planning software loading strategies.

This article is intended for designers and system administrators alike who need to know what the requirements are for a commercial network management tool capable of delivering software automatically in a heterogeneous environment.

Introduction

The Open-System trend has brought to large organizations a workstation landscape that looks like the rolling hills of Virginia in autumn. Unfortunately, we, the system administrators, have to feed "our trees" every year. It has become a nightmare to install a new operating system or application software on hundreds of workstations, often unlike versions at the same time on different workstations.

Moreover, the new software has to be compatible with the existing one and thus the task to make up the list of who has what and who gets what becomes even more complex and tedious. There is hardly ever enough personnel available for testing and upgrading software but even if there is, the problem remains to get all systems configured consistently within a reasonable amount of time because of errors made by the system administrators due to the extreme monotony of the loading and installation process.

Moreover, the new software has to be compatible with the existing one and thus the task to make up the list of who has what and who gets what becomes even more complex and tedious. There is hardly ever enough personnel available for testing and upgrading software but even if there is, the problem remains to get all systems configured consistently within a reasonable amount of time because of errors made by the system administrators due to the extreme monotony of the loading and installation process. Moreover, the new software has to be compatible with the existing one and thus the task to make up the list of who has what and who gets what becomes even more complex and tedious. There is hardly ever enough personnel available for testing and upgrading software but even if there is, the problem remains to get all systems configured consistently within a reasonable amount of time because of errors made by the system administrators due to the extreme monotony of the loading and installation process.

The vendors supply us with nice tools but unfortunately, almost everyone has his own system which is different enough to force a learning process on those who have to install software. Here we encounter the first problem when we try to automate the software loading process: once the files are successfully transferred to the target machine, the install procedures of that particular type of system have to be used. Another problem occurs with in-house software packages. They have to be packaged into the vendor's format so they can be recognized by the system as loaded software. In order to understand the problems involved we list the requirements to create and manage a system which loads software automatically. We will describe the system *AUTOLOAD* which has been developed and commercially installed by the authors, focusing on its architecture and functionality. Also, the pro and cons of its implementation in C++ will be treated.

Requirements and Problems

The procedure of interactively loading and installing software packages onto a workstation is rather complex. In order to automate this procedure, one must think about several important constraints before designing such a system; requirements regarding the data which are necessary for loading, the control of the loading procedure and implementation issues.

The following prerequisites are necessary to allow a unique handling of all software packages

which can then be performed automatically:

Software Inventory The system has to know what software is presently on any particular target, for example for upgrades, and what is intended to be on there.

SW Package Format In-house developed software has to be brought into the vendor's format so that their SW delivery tools can be used. Each system requires its own handling of installation procedures.

Software Dependency A lot of software packages depend on the existence of other packages. They either will have to be loaded too, or the dependent software exists already on the target machine. Moreover, the loading order has also to be managed according to its priority.

Removing Software Packages Removing SW packages remotely requires not only a *very careful* implementation but also rigorous checks of *each* uninstall script before depending on it.

A number of additional considerations essential to loading and installing software *automatically, in parallel and unattendedly* are now explored.

One network management system should be able to service *all* UNIX workstations within the organization. Workstations of various vendors, often with different architectures and versions of one supplier, e.g., Sun-2, Sun-3, Sun-Sparc have to be serviced distinctively. Furthermore, groups of workstations with the same hardware-equipment cannot be treated as identical when considering their software inventory. Therefore, a major requirement is keeping the *autonomy* of each workstation, especially allowing the user to install und uninstall SW on its own and yet be able to register this fact.

Replacement of User Interaction A number of SW installations require answers by the user when loading. They need to be replaced by executable scripts, e.g., shell ''here''-documents.

Asynchronous Procedure If many workstations are to be loaded at the same time, the procedure can only be asynchronous and is thus a very difficult task to master: an enormous amount of return messages have to be managed and partly acted upon.

Installing the System from Scratch UNIX System V has limited capabilities of network booting, if any at all. Bringing up a system from scratch over the net is by no means a simple task. This applies to new systems, system crashes and to the case when the entire network system has come down with a virus attack, in which case the backups are useless for automatical reloading. Software from backups must be individually examined after a virus attack before it can be used again.

Loading Success Suppose the machine has been loaded automatically. How can be determined whether or not this was entirely successful when everything happens remote?

Locks while Loading It may happen that a package which is supposed to be loaded automatically cannot be installed because parts of the package are active (i.e., one of the programs to be loaded is running). Under these circumstances the user might have to be logged out and the machine to be brought into another run-level.

When unattended loading and installation of software is performed, one has to ensure that the software package to be delivered and installed is available during execution time. Therefore, removable media like tapes or CD's are not acceptable, for who can guarantee that someone does not tamper with the CD while the SW is being loaded? Therefore, the authors require that all software packages reside on the hard-disks of one or more specific workstations, called SDNs (*Sofware Delivery Nodes*). If the net is very large, it is advisable to have at least one SDN in each subnet. From there the data of each software package is transferred to the workstation to be serviced.

In addition to the requirements discussed above, a number of decisions concerning the implementation of such a system have to be considered:

Display If a graphical user interface is chosen, it must be designed in such a way that all operations can be accomplished easily by using the mouse. Furthermore, also the output of the system in this case must be represented in a graphical way. A crucial requirement is that the operation of the user interface must be independent from the rest of the system so that the user interface is never blocked by any operation of the loading process. Therefore, means for communication between the user interface and the application, e.g., by message passing, must be designed.

A system for automatic, unattended loading of software must be modified for each workstation type, adapted to its particular SW delivery technique. Furthermore, the master program with the user interface should run on a variety of different workstations. To keep one source for all platforms is difficult, particularly for the differences between System V and Berkeley Unix. In the future, the system will have to be compatible with OSF's DME (Distributed Management Environment) tools.

Software Transfer Level The system can use the r-commands and thus function on the top (shell) level. A low-level system requires the implementation of a network daemon either with intermediate tools such as RPC (Remote

Procedure Calls) or low-level ethernet TCP/IP (Transmission Control Protocol/Internet Protocol) or UDP (User Datagram Protocol) functions. In order to avoid lengthy loading times, a protocol with a high data transfer rate, but with error detection or correction is to be selected.

Accessibility A system based on remote logins will fail when the target machine has a limited user license and the present user exhausted it.

Security Using the high-level approach requires some kind of superuser privileges of the target machine. This means special precautions have to be taken.

Available Tools

UNIX system managers will need a good understanding of the major network management products and their limitations as to what extend they really solve problems like loading software automatically to enable effective decision making. A number of tools exist which provide an underlying structure for the development of software distribution applications, but they are usually directed towards a particular platform. OSF plans to address an open environment by having the Distributed Management Environment (DME) developed. Nevertheless, a solution for a heterogeneous environment simply must master machine-dependent tools like *inst* of SGI, *newprod* of Intergraph, *extract_unbundled* of Sun, *setld* of DEC, *installp* of IBM and so on, as well as the three UNIX systems: System V, Berkeley and OSF.

Many people have something "in store" to automate the tedious process of upgrading software and rely on some tools either found in public domain or supplied by their vendor who said: "You can use this, but you are on your own." Is it worth to continue in the direction started, or has the time come to reconcile?

Levels of Approaches

When we have a closer look at the tools for loading and installation of software packages as provided by hardware and software vendors, we easily detect two classes: tools for *local installation* and tools for *network installation*. In the first case, the interactive control is to be performed on the workstation's "console" and the distribution medium has to be in place in a drive which is directly connected to that workstation.

Network installation tools allow the package to be loaded from some other machine on a local area network. In some cases, even the *control* of the loading and installation process can be performed remotely.

The remote access of data, i.e., copying data from one workstation to another and executing commands remotely can be accomplished on two levels: on a high level using r-commands (e.g., *rcp*) or similar commands, or a low level, on which the communication and the handshake are programmed individually. Although the first variant is much easier to be implemented – you just write down a shell script – it bears several severe problems. Beside a lower performance, the source and exact kind of errors can almost never be detected when r-commands are used. Just think of the error message "permission denied" when using the *rcp* command. A further problem occurs when r-commands are used: the system has to obtain root-privileges on each target workstation. This may lead to a serious security gap.

Example of a Solution: AUTOLOAD

The authors have implemented a network-based SW delivery system taking the low- and high-level network approach. The system is called *AUTOLOAD*. The high (shell) level was instituted first and found to be too vendor-specific and not practical for the unattended loading of a large number of workstations. Nevertheless, it is being kept alive along with the low-level implementation because it continuously serves as a prototype and is a convenient means to establish programs automatically which substitute user interaction when executing install scripts.

Functionality of AUTOLOAD

AUTOLOAD is designed to allow the *unattended* loading and installation of software packages onto several workstations *simultaneously*. A graphical user interface allows the user to select the software packages and the workstations to be serviced in a variety of ways: any group of workstations can be

- loaded with any set of software packages belonging to a selected release;
- restored from scratch as they are intended to be configured or as they were before, e.g., for new installation or after a virus attack;
- loaded with default software;
- loaded with basic software packages belonging to a particular type of workstation;
- loaded with updates of selected software packages (to workstations which hold an older version of that software package).

Furthermore, software packages can be removed from a workstation again. Here one can

- remove one software package from a group of workstations which have this package installed, or one can
- remove a set of software packages from one workstation.

All these different selection modes can be combined into "loading jobs" which can be started at an arbitrary time. This is extremely useful, because then loading actions can be carried out when the network load is low, e.g., during night hours or the weekend.

Of course, loading actions in progress can be interrupted properly, i.e. the actions in progress are completed before *AUTOLOAD* terminates. Further functions allow to

- show the current status of the loading activities in a graphical way, and to
- perform several additional management functions (e.g., browsing the data base).

Figure 1 shows the software selection mask of the graphical user interface of *AUTOLOAD*.

The Layout of AUTOLOAD

The requirements for unattended loading and the functionality of the system as described above led to a layout of *AUTOLOAD* consisting of:

- a graphical user interface GUI,
- a data base for keeping all data pertaining to the state of all workstations and all software packages,
- the *AUTOLOAD* -kernel which processes and controls all user actions, and
- the *AUTOLOAD* -daemon which is in charge of handling the communication over the network.

The next sections describe the four major parts of *AUTOLOAD*.

Contents and Structure of the Data Base

When *AUTOLOAD* wants to load or remove packages from workstations, a lot of information about the workstation and the software package are needed, e.g., the internet address of the workstation, or on which software delivery node the package to be loaded resides. Therefore, a *data base* which we call *ALDB* is one of the key parts of *AUTOLOAD*. The *ALDB* is organized as a relational data base and uses an ASCII representation of its data records (like the file "/etc/passwd"). The *ALDB* holds all necessary information about workstations, software packages, and software releases.

The data stored for each **workstation** consist of the following four groups:

Workstation Identification A workstation is identified uniquely by its name and internet address.

Administration Data in this group are, e.g., the location of the workstation and the name of its administrator.

Hardware Equipment Stored data, e.g., the type of the workstation, the size of its memory, the hard-disks and special hardware are connected to it.

Software Equipment This group contains all data used in connection with the software installable or installed on a workstation and will be described in detail below.

The data stored for each **software package** can be structured into groups as well:

Software Identification This group contains the data to uniquely identify a software package, e.g., its name, version number, date and the release it belongs to.

Loading Prerequisites This group contains information about prerequisites which must be fulfilled in order to load the software package successfully. Typical entries in this group are the required size on the file-system partitions, the workstation model (e.g., Sun3 or SunSparc) and necessary hardware, e.g., a color graphic equipment. Very often, software packages depend on other software packages, e.g., windows applications need the windows basic system in order to run. Within the *ALDB* a relation defines which software packages depend on one another.

Loading Control Information The information in this group is needed by *AUTOLOAD* to deliver and install software packages, e.g., the location of the software delivery node on which the

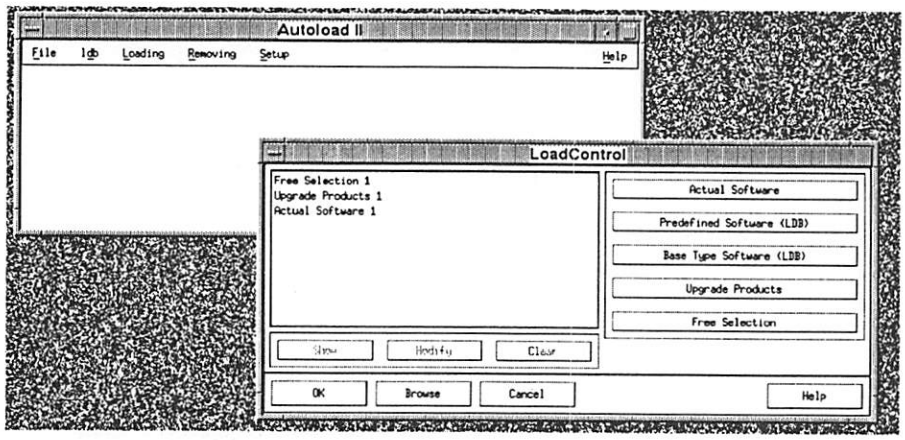


Figure 1: Load Selection

software resides, or the name and residence of the install script.

All software is grouped into **Release Levels** to allow a more convenient selection of software to be loaded. Every software package belongs to at least one release of which it is known to be compatible with. In many cases, the release levels will be the same as the one of the operating system, but *AUTOLOAD* allows to define arbitrary releases. This feature facilitates testing and integration of new software packages: one may define a test-release to have it turned later into a production release level.

One of the most important features of *AUTOLOAD* is that it keeps the *autonomy* of each workstation. This means, that each workstation serviced by *AUTOLOAD* may be configured individually. In practice, however, this may lead to a substantial administrative overhead, especially in an environment with a large number of workstations (e.g., thousands of them). One may detect that there are a number of groups of workstations which are configured quite *similarly* to each other. A well-known example for this phenomenon is that there exist ''developer''- or ''CAD''-workstations, or the ''management-computer''. Whereas some network management systems can handle this situation only by treating all workstations of one group alike, *AUTOLOAD* has a built-in mechanism to model the notion of ''*similar* configurations'': subset of software packages which are installed on each machine. A hypothetical workstation with exactly this selection of software packages is called an ''Abstract Workstation''. *AUTOLOAD* allows the user to define an arbitrary number of abstract workstations. Each ''real'' workstation then belongs to some abstract workstation. To reflect the *similarity*, a list of software to be loaded *additionally* is kept in the data base. Thus the software to be loaded on a given workstation per default is the software of its abstract machine (the ''Base Type Software'') *plus* the additional software.

In the *ALDB* we additionally have to keep information about which software packages are *actually* installed on a given workstation. This list may differ from the default list, e.g., if the local administrator decided to remove some packages in order to obtain further space on the disk. The information must also be kept in the *ALDB* because it is needed to restore the workstation.

In conclusion, we have four distinct types of software sets belonging to one and the same workstation:

Base Type Software This is the software which belongs to the workstation's ''Abstract Workstation'' type described above.

Additional Software This software is to be loaded *in addition* to the base type software. It is especially useful when the workstation data is entered into the *ALDB* : once the basic type is chosen, the configuration is entered automatically and the rest is easily ''picked'' by point-and-click.

Actual Software This list of software packages is obtained by ''visiting'' the workstation and checking which software packages are really installed, regardless of whether they had been loaded by *AUTOLOAD* or by hand.

Software Loaded by *AUTOLOAD* This list of software packages reflects the software packages as processed by *AUTOLOAD*. It is used to fully restore a workstation[1]

In the ideal case, the last two lists are identical. It is one of the tasks of the network administrator to have a close look at the difference between the two lists, since they show which packages have been installed or removed without knowledge of the *AUTOLOAD* -administrator. They even might have been installed illegally.

System Architecture of AUTOLOAD

The system architecture of *AUTOLOAD* has to reflect the requirements as stated above, namely the asynchronous operation and the simultaneous servicing of many workstations. This results in a system architecture of *AUTOLOAD* as shown in Figure 2.

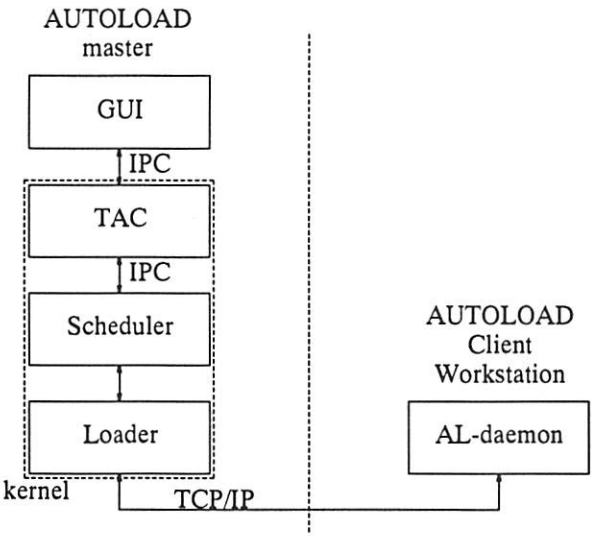


Figure 2: System Architecture of *AUTOLOAD*

On each of the client workstations, i.e., the workstations which are to be serviced, a *daemon*, the *AUTOLOAD*-daemon is running. It communicates with the *AUTOLOAD*-master (left hand side of the figure) using TCP/IP and remote procedure calls (RPC).

[1] *AUTOLOAD* must use this list instead of the **Actual Software**, since the latter list may contain software packages which may not be loadable by *AUTOLOAD*

We will now focus on the architecture of the *AUTOLOAD*-master. The three main parts of *AUTOLOAD* are the graphical user interface GUI, the transaction controller TAC, and the SCHEDULER/LOADER(S) which are separate UNIX processes communicating with each other using interprocess communication (IPC) message passing.

For each workstation to be serviced (loading, removing, visiting), a new LOADER process is spawned off which is in charge of servicing this workstation. Therefore, a larger number of workstations can be handled simultaneously. However, care must be taken, not to overburden the network. To avoid this, the user can limit of the number workstations to be serviced simultaneously.

The graphical user interface GUI is a X-Window-based user interface with the MOTIF window manager, but others including Sun's Open View work just as well. All operations can be directed with the mouse. Furthermore, error messages, interactive queries, and the current status of the workstations which are being serviced at the moment are displayed (see Figure 1). The GUI only communicates with the TAC.

The transaction controller TAC is in charge of synchronizing all scheduling and loading actions. It communicates with the GUI and receives the user commands from there. These commands are interpreted and transformed into commands for the SCHEDULER of the respective workstation. The TAC also contains the interface to the data base *ALDB*. The TAC being the central part of *AUTOLOAD* creates a transaction log-file which is used to generate reports about present and past loading actions. For each workstation to be serviced, there exists one process consisting of the SCHEDULER and the LOADER. When a loading job is started, the list of software packages to be loaded is received from the TAC. The SCHEDULER transforms this list into a *loading schedule*, taking into account the dependent software of the given software packages and the loading priority. If dependent software is not already installed on the workstation, it will be loaded by default. The SCHEDULER then activates the LOADER to load one software package after the other. In order to do so, it has to communicate with the *AUTOLOAD*-daemon on the software delivery node and on the workstation.

Errors which may occur during the loding process are subdivided into three classes: *fatal* errors, *non-fatal* errors and *retry* errors. Whereas in the first case the entire loading procedure for a workstation is cancelled (e.g., when the network of the workstation is not reachable), only the loading of the current software package is aborted in the second case. After that, a rescheduling takes place and the SCHEDULER tries to load and install the remaining software packages as far as possible. In any case, the software dependencies and the loading order must be obeyed. *Retry* errors are mainly due to time-outs. This is a common mechanism in network programming: to avoid a deadlock, an operation must be completed within a given time, which can be set externally in *AUTOLOAD*. In such a case, a number of retries are allowed before the loading of a software package or of the workstation is aborted.

Implementation of AUTOLOAD

Object-Oriented Implementation

Except for the database which is in C, everything of *AUTOLOAD* was coded in C++. The Graphical User Interface (GUI) was implemented in X-Windows, using for parts of it MOTIF's user interface language (*uil*). Doing this together with C++ was not easy and possibly not worth the effort. The GUI is event-driven. The handling of messages between the TAC and the GUI is kept transparent with respect to the objects involved. Each object of the TAC that wants to display something sends a message to the main object of the GUI. This object in turn decodes a part of this message and sends it to the particular display object. Control information added to the message ensures that the receiving object always knows who sent the message.

The central GUI-object is designed in such a way that all display objects (e.g., error windows, display browsers) can be re-used.

The network communication is programmed in RPC (Remote Procedure Calls) with the server, client daemon and utilities programmed in C++.

The ASCII database *ALDB* was written so that it can be easily ported to any other platform. Nor does a license have to be obtained for it. Furthermore, all data used by the *ALDB* are human-readable. Just think of it – what would we do if we couldn't read the *inittab, passwd* or the *rc-files*!

Documentation

Maintenance of such a complex system is a major problem. If changes are to be made, they are generally only within one class and thus don't affect the rest of the system unless they were made in the interfaces. An extraordinary effort was made to keep the design and implementation document complete and up-to-date – it now consists of 224 pages. Not only are the classes very meticulously described in this document, but each class has again a description which can be automatically extracted to an on-line manpage in standard UNIX format. In-line documentation is likewise very extensive.

Porting AUTOLOAD

The Loader

The procedure of loading and installing a software package onto a given system differs from one UNIX-system to the other. Some of its

mechanisms have been described above. The LOADER of *AUTOLOAD* for a given type of workstation has to control (and in some cases even to simulate) the loading and installation process. Therefore, the LOADER has to be a different one for each type of workstation. Nevertheless, the loading and installation procedures are quite similar to each other and consist of the following major steps:

Check Installability Checks have to be made if the particular software package can be loaded and installed at all.

Transfer of SW from SDN to Target The data of the package (archive and control files) has to be transferred from the distribution medium or the disk of the software delivery node onto the workstation.

Manage Install Scripts A package-specific script may have to be invoked, called the *install script* which:

- asks several configuration questions
- unpacks the archive and copies the files to their final destination
- modifies system configuration files and changes permissions

This install script is delivered together with the software package and is left unchanged by *AUTOLOAD*. All *AUTOLOAD* has to do is provide the correct calling environment and the right answers.

Postinstallation It may be necessary to invoke a package-specific script to do additional work, the post-install script.

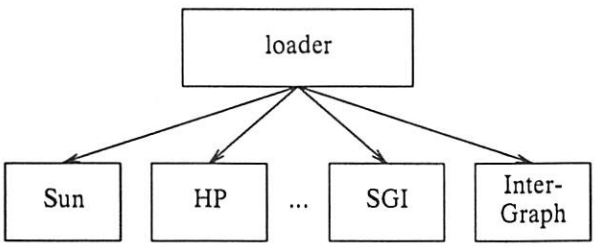


Figure 3: Class Hierarchy of the Loader

The object-oriented design of *AUTOLOAD* facilitated the adaptation of the LOADER to different workstation models to a large extend. As shown in Figure "The Loader", we use C++ class inheritance together with virtual functions to perform the adaptation. The base class is a *generic* loader which contains a skeleton of the loading and installation procedure as sketched above. Furthermore, it implements a large number of utility functions which are needed by this procedure. They are defined as *virtual* functions.

When a LOADER for a specific workstation model is to be defined, i.e., a new platform has to be created, it is derived from the generic loader. Differences in the loading and installation procedure can then be easily accomplished by adding or modifying virtual functions in the derived class. Experience showed that for adapting a loader, only about one to five of these auxiliary functions had to be adapted in the derived class.

Porting

All parts of *AUTOLOAD* have been designed in a highly portable way. For basic data structures and algorithms we used the NIH C++ class library. The NIH library has good portability except for the lightweight processes which for that reason were not used. The error and exception handler class was also not used, because we preferred to write our own. The C++ compiler being used is the AT&T Translator Version 2.1. in source and ported to each particular platform. Its port to other platforms in some cases needs major adaptations with respect to the handling of the variable argument list. Furthermore, some C compilers (and the C preprocessors) have problems in processing the C-code as generated by the translator, mainly because of its size and complexity.

The MOTIF user interface provided more difficult problems, especially the migration from version 1.0 to 1.1. Even severe bugs were found in some ports of the MOTIF development kit.

Conclusions

System administration software should be developed so that it is simple, uniform and fun to use. Simplicity comes with having no longer to type in information the system knows anyway, e.g., about its own constellation or any of its addresses. Uniformity lies in not having to learn what kind of software loading mechanism has to be used for which system. How to make it even simpler? Automating the task of loading of upgrades *for all systems* is one step in the right direction. OPEN SYSTEMS are not supposed to mean MORE WORK.

As described, loading software at work while sitting at home requires a very complicated system with an intelligent database. The database has to be kept up-to-date and – what is even worse – "completed" with information about all workstations in the first place. This requires an enormous organizational effort which is not to be underestimated. One of the most burdensome tasks in developing *AUTOLOAD* was to get its users to set up their company in such a way that all requirements for automation are met, e.g., to enter the data into the database, to establish integrity and consistency of the data, to package in-house developed software. We supplied a lot of tools for this and don't even want to think about where we would be now had we not done this. One of the tools reads existing configurations and enters the information automatically into the *ALDB* database.

What could be fun about loading software? Just watch the display bars being filled as one package after another is loaded onto many workstations at the same time. Instead of having to answer questions, the system does it for you and the best part of it is that you can leave it.

At present, *AUTOLOAD* serves workstations on five different UNIX platforms. Loading a particular workstation by hand and using a tape device takes five hours, using the network and *AUTOLOAD*, it takes one hour. Taking seven subnets and loading 10 to 15 WS's of many different makes simultaneously with a software delivery node in each subnet, is yet to be tested – the need has thus far not arisen – and represents the ultimate goal.

Author Information

Dieter Pukatzki is employed at Leading Edge Technology Transfer at Kurt-Huber-Str. 11, D-8032 Muenchen-Graefelfing, Germany. Reach him via telephone at (49) 89 - 85 21 82 or fax at (49) 89 - 854 38 68. His electronic mail address is dieter@guug.guug.de .

Johann Schumann toils at the Institut für Informatik division of Technische Universität in München, Germany. Reach him electronically at schumann@informatik.tu-muenchen.de

ipasswd – Proactive Password Security

Jarkko Hietaniemi – Helsinki University of Technology

ABSTRACT

Recently efficient and easy-to-use implementations of dictionary-based password cracking tools have been made publicly available. Armed with intelligent guessing strategies and fast crypt(3)-functions, these tools enable anyone with a C compiler and some CPU time to burn, to pry open a considerable proportion of the passwords in standard UNIX systems. The origin of this problem are the poor passwords chosen by the users. This paper discusses the problems involved in password security and the techniques used to solve these. In particular, one counterattack to this problem, *ipasswd*, is described. *ipasswd* is a proactive or preventive security system; it tries to prevent the inadequate passwords ever reaching the system databases. This paper presents the techniques used in a prototype implementation of *ipasswd* to test, with great suspicion[1] the passwords supplied by the users.

Motivation

Passwords are the usual form of authentication keys to multiuser computer systems. Together with the account name they are requested by the system, provided by the user and verified by the system. After that point, every resource allocated to that particular user is freely available. Inside the multiuser system access to various resources is regulated by many different methods: in UNIX these are for example process memory limits, disk quota, network service points, and especially the file protection scheme. These can be used by the administration and by the users themselves to achieve a fine-grained and multi-level protection whereas the first gate, the primary access to the system, is protected by a few simple characters, the password.

What Makes or Breaks a Password

The passwords are stored in UNIX systems in an encrypted form which is one-way. The encryption cannot be reversed to obtain the original, clear-text password. This does not, however, rule out the possibility of simply guessing. When guessing passwords, the law of least resistance is well kept in mind:

The Law of Least Resistance

People tend to use three principles in selecting passwords:

- ignorance
- laziness
- arrogance

All these result from the partial, or as in the usual case, complete, lack of user education.

In the classic study on the area of password security [Morris&Thompson79] users' passwords were probed and classified before encryption and the results were disheartening: 86% of about 3300 passwords checked were of the most simple classes: very short or from very small character classes like single cased or directly from some dictionary. With the current hardware and software available, passwords of this sort can be cracked in a matter of days. This method of monitoring the password quality is *proactive* or *preventive*, it is conducted before the encryption. The cracker's parallel to this is a Trojan horse impersonating a legitimate login procedure and relaying the entered passwords to the evil-doer [Ritchie79].

The counterpart method to proactive checking is *retroactive*, or *reactive*, worrying afterwards. In a paper [Klein90] a thorough attack on 15000 encrypted passwords is described. The results were again appalling: 24% of the passwords were guessed correctly. This test was purely dictionary based. Intelligent methods were used in both selecting the trial order of various dictionaries and in mutating the words. The most fruitful dictionary categories were users' real and account names and common first names.

Both these studies were done with English dictionaries. In the spring of 1991 a test similar to Klein's was done at Helsinki University of Technology, Finland (HUT). A large Finnish dictionary[2] was available and 16000 passwords were checked with various mutations like altERnAting casing and reversal but the mutations used were not as comprehensive as Klein's and the complex

[1] *ipasswd* stands for "inquisitor passwd". Webster's Encyclopedic Unabridged Dictionary of the English Language:

inquisition n. 1. an official investigation, esp. one of a political and religious nature, characterized by lack of regard for individual rights, prejudice on the part of the examiners, and recklessly cruel punishments. 2. any harsh or prolonged questioning. ...

inquisitor n. 1. one who makes inquisition. 2. a questioner, esp. an unduly curious one. ...

[2] 265368 words or 3175636 bytes, with a hint of Swedish thrown in; Finland is a bilingual country.

inflectional system of Finnish makes guessing harder. Therefore only 1700 or 11% of the passwords were guessed: of these 56% were Finnish or Swedish, 26% English and the rest 17% were in other languages, most notably German or French, or unclassifiable, like "qwerty".

It is unlikely that users have grown much wiser in selecting their passwords since Morris' and Thompson's study but the technological advances alone would make void any user education made since those days. The threat of brute force guessing has increased because of three developments:

- Increased CPU speed: chip technology has made giant leaps since UNIX's childhood days
- New fast crypt() implementations: several speeded-up versions of the encryption algorithm itself have been made publicly available. Some of these are:
  - fcrypt() by Robert Baldwin, Icarus Sparry, and Alec Muffett
  - ufcrypt() by Michael Glad

  Both of these are easily available with the **crack** program [Muffett92].

  These two advances alone turn a year into a working day: the speedup from μVAX-II[3] and the original algorithm to HP9000s720 and ufcrypt() is approximately thousandfold.
- Intelligent guessing strategies: current cracking tools do not blindly guess words from the beginning of /usr/dict/words. Instead, they use sophisticated knowledge of bad passwords to make the right guesses as quickly as possible. They first check the account name itself and then exploit user information stored in the GCOS field of /etc/passwd. These are by far the most profitable guesses.

The cryptological strength of the crypt() function is a fascinating mathematical problem but the interest remains purely academic as long as the users use their own account name as their password.[4]

The UNIX systems administered by the Computing Center HUT have about 8100 accounts owned by about 7200 different users. This large user population makes thorough and frequent retroactive password checking unfeasible, even if only the changed passwords are checked. This depends on the definitions *"thorough"* and *"frequent"* but automatic retroactive password checking on heterogeneous UNIX systems is a waste of both administrative work and computing resources. Moreover, an automatic transfer of encrypted passwords over the network with either TCP/IP or distributed filesystems is a security hazard because of network security. The whole idea of retroactive password checking is to be considered wasteful. If the crackers do it, they endanger system security and waste system resources, most notably CPU capacity; if the administrators do it, they enhance system security[5] and waste system resources. The only justifiable use for retroactive checking is purging the system clean of bad passwords as the final step of creating a safe password environment.

[3]Morris and Thompson did not use the current DES-based algorithm in their PDP-11 but rather a U.S. Army World War II encryption technique. No performance measurement of the current crypt() algorithm on the PDP-11 is available.

[4]There is a rumor that most multiuser systems contain at least one account which has as its password its account name. Accounts like these are called *joes* [Garfinkel&Spafford91, Brand90].

User Education

Left to their own ways, some people will still use cute doggie names as passwords [Grampp&Morris84]

By far the most effective solution is user education. Very few administrative security efforts can achieve as much as a security-aware, well-educated, co-operating user community. There are certain potentially problematic user populations:

- Large and heterogeneous user population with high turnover (academic sites). Close co-operation with the student registration staff is needed to remove graduates that no longer need their undergraduate accounts. The same problem applies to guest researchers.
- Reluctant, even hostile, attitudes towards computers (mostly non-computer commercial sites)

The problem of the first population is the large number of people and the inertia involved. If many thousands of people are to be informed of sound password selection policies, some kind of urgency sorting must be done, the more *"valuable"* account holders must be trained first. The problem of the second population is an active ignorance of the computer systems and the security measures required therein. These two user population models are not mutually exclusive, an academic site normally contains both computer and non-computer students and normal staff. People are generally well aware of personal computers and their use but the need for co-operation in multiuser systems is often poorly understood.

Some Problems

Unused Accounts

Perhaps the greatest single security risk is an unused or dormant account. The rightful owner either uses the account very infrequently, not at all, or does not have knowledge enough to notice that his account has been tampered with. Therefore the

[5]But *very* slowly compared to proactive methods.

crackers have a strong incentive to crack open the account and revitalize it to use it as a stepping stone to further conquests [Stoll89]. Thus it is of paramount importance to at least close infrequently used accounts. These should be in due time removed from the system both because they endanger system integrity and waste resources, normally disk space. An unused and closed account should not be enabled before a personal visit to the administration.

Group Accounts

The question of group or shared accounts is of course a site-specific policy decision but in general they are undesirable for two reasons. Group's work may get corrupted when many members of the same group try to work simultaneously. The security is of suspect value because many different people know the password. Therefore in a case of misuse pinpointing the real culprit is difficult. The rights of such shared accounts are best kept rigorously limited to the bare essentials required for the particular project. The account should be closed and if possible, removed, from the system as soon as possible.

Another additional danger is that some people may add the group account to their .rhosts thus opening brand new holes.

Guest or Demo Accounts

Guest or demonstration accounts are akin to both unused accounts and group accounts. They may lie dormant for a long time, their users are not aware of each others' activities and the password may be known to many people. The need for guest or demo accounts need to be weighed carefully and the rights of such accounts must be restricted.

Enforced Changing

People may react sometimes aggressively to the complaint made by the administration that their password is inadequate and needs changing. Personally they may not have anything of importance in the particular system but they do not realize that by leaving just one foothold wide enough for the crackers to edge in, they are risking the integrity of the whole system [Klein90]. When or if they finally change their password, the new one is most probably as bad or even worse than the original. This is almost inevitable because the change is made involuntarily and normally in haste, without proper consideration and careful thought.

Password Aging

For the very same reason, timeout of unchanged passwords or password aging is not altogether a good idea [Grampp&Morris84]. The idea behind password aging is to narrow the window where a cracked password can be used. The window required is extremely narrow: modern tools can crack a password in a matter of days, even hours. If users are required to change their passwords too frequently, the most probable scheme they will apply [Klein90] is to use some appallingly simple short password and prepend the month of change to it, for example, joejan, joefeb, joemar, ... or if numbers are required in the password and the *de facto* minimum length of six is not asserted: joe1, joe2, joe3, ... Password aging can be used beneficially but careful thought should be applied, the frequency depending on the site policy. Observing the law of least resistance, one can readily see that people may obediently change their password as required on the day required but they will change back to their old trustworthy, easy-to-type, easy-to-remember, maybe even to the password written on a note attached to their terminal: joe123, joe456, joe123, joe456, ... To prevent this toggling back and forth, some systems define the *minimum* time required between password changes. This idea is not much better: it prevents legitimate quick password changes when for example the user realizes that the new password was perhaps too easy after all. A better solution would be to keep log of each account's passwords in their encrypted form and check a new password candidate against this history.

Password aging can be used beneficially in detecting unused accounts. However, more sophisticated methods and policies should be used to determine the activity of an account rather than just monitoring the password change frequency.

Special Purpose Hardware

Special hardware can be used in addition to the password. This is the most efficient security measure available[6] and also the most difficult, expensive and inconvenient alternative. These measures contain various keys, cards, voice, signature, fingerprint and retina scanners. The keys can be generated just for a single session. Because of the required complex technical equipment and the additional burden placed upon the users this method is quite out of the question except for high security installations like commercial research laboratories, military and intelligence agencies. [Brand90]

An Overreaction

There is quite wide-spread variant of security by obscurity in use. Some sites have disabled the finger service because they do not want outsiders to retrieve account names or real name information from the systems to be used as material for dictionary-based cracking attempts.[7] This train of thought is somewhat misguided: if the principle were

[6] Short of cutting off the system from the network and setting an armed guard behind each terminal.

[7] Partly this may be left-over paranoia caused by the Internet Worm. One of the major wormholes in VAX-platforms was fingerd [Spafford88]. Public fixes for this have been available almost from the beginning but the bad news may be traveling faster than the good news.

followed literally, electronic mail and various conferencing and news services should be cut off because they also carry name information relating to users. Comprehensive lists of both real and account names are easily obtained without any kind of information flow out of the systems so jealously guarded. The method resembles putting the lights out, sitting in the dark and pretending that no one outside will notice the house. This could be called *retroretroactive*: after realizing the possible weak condition of their users' passwords, the administration does not enhance password security except by inconveniencing the network community.

Credo

If the users are ignorant enough to use their account names as their passwords the administration should be enlightened enough to

- Organize user education on password security
- Install proactive procedures in their systems to prevent further bad passwords
- Apply retroactive tools to their own systems to dig out bad passwords

Prior Art: Still Some Problems

Vendor-supplied

The vendors have tools for easing the situation. The main problem with vendor-supplied methods is that they are turned off by default when shipped and they are by definition limited to their architectural platform. The lack of the source code restricts site-dependent enhancements and expansions. Sometimes the implementation may contain a bug or a feature that lessens security: Berkeley version of passwd(1) insists on the minimum length of five characters for new passwords *but* this can be circumvented: it gives up on the third try.

All the password quality checks are built in and are not configurable. In some UNIX systems a more flexible system is available: for example AIX, SunOS and Ultrix give the administration the possibility of configuring the strictness of password checks on a site-by-site and machine-by-machine basis. These checks try to improve the password quality by enforcing:

- Minimum length: the minimum length required that gives some kind of barrier against today's cracking tools is six characters
- Character set richness: not single-case, not just alphabetics
- Enough difference between the old and the new password in terms of changed characters
- The maximum number of times a single character can be repeated in a password

The main cause of inadequate passwords, the use of simple names and words from easily available dictionaries, remains.

Shadow Passwords

There is one vendor solution that enhances password security remarkably: the use of *shadow* passwords where the encrypted passwords are hidden from the ordinary user in a directory or file called something like /etc/security or /etc/shadow. This prevents the crackers from getting the encrypted passwords to crack but does not totally solve the problem. The law of least resistance: people are prone to use the same password in many systems and the encrypted passwords are not necessarily hidden in all of them. Because passwords travel in the network, eavesdroppers can again strike and gain access to many systems at the same time. One solution to this problem is the MIT Athena Kerberos.

Same Password, Different Accounts

People may have accounts in multiple systems and the law of least resistance says that they are going to use fewer passwords than there are systems. Because of this the cracker may get hold of many accounts simultaneously but checking this when a password is changed is very difficult because the systems in question may reside in different administrative domains and they may have very different UNIX versions. An even worse situation may be at hand when a privileged user uses the same password for his administrative duties and unclassified work [Brand90].

Generated Passwords

Some vendors give the user a possibility to use automatically generated passwords which are assembled for example from "*easily pronounceable and thus easy to remember*[8] syllables. For one thing, they are not easily pronounceable even for English speakers because they are random and difficult to associate with any familiar word; for non-English speakers they are quite simply awful. Because of their randomness they are difficult to remember and people are likely to write them down and thus weaken the password security.

The *pass phrase method*[9] [Brand90] is feasible to implement but it is much easier and more secure to teach the method directly to the users.

The worst thing about "randomly" generated passwords is that they may not be random enough: the randomness algorithm may be deterministic which makes the cracker's job very easy indeed [Morris&Thompson79]. The other generating

[8]That is how the vendors' documentation calls these passwords.

[9]Sentences of an approximate form *adjective(s)?-noun-verb-adverb* where all the components are chosen randomly. The password is made up selected characters of the components. For example by selecting the second characters of *"colorless green ideas sleep furiously"* one gets *"ordlu"*. This makes a good basis for a password but more variation like digits still must be added.

methods like totally random character sequences and pure numbers are considered inadequate for the reasons already discussed.

Writing Down Passwords

The normal security measure is to tell the users not to write down passwords. This should not be categorical and absolute: the rule should be:

> *Do not write down your password on-line or near to your personal terminal.*

If people keep their password in their wallet or purse or at home in a drawer or strongbox and the password gets stolen, they will usually have worse security problems than their passwords.

Freely Available

The vendor solutions are currently mostly proactive. Both proactive and retroactive tools are freely available in source code form.

Proactive

npasswd

npasswd is a replacement for passwd(1) for BSD4.3- or SunOS 4.0-like environments. The checks made are very much like those in the vendor-supplied solutions but they contain a possibility to check for an exact match against some dictionary file [Hoover92].

Shadow Login Suite

shadow login suite is a replacement for login(1) and all the other account-related tasks for SysV-like environments, the BSD port is not yet complete as of the time of writing this paper. The basic checks are a little better than those of *npasswd* but they do not contain any dictionary checks. By default only the minimum length is checked. [Haugh92]

passwd+

passwd+ is a replacement for passwd(1). Its tests are extensive: they contain exact and pattern matching against dictionary files, program outputs, GCOS checks and logical expressions for combining the tests. *passwd+* tries to be UNIX version independent and it is highly configurable. [Bishop92]

perl Rewrite From the Camel Book

In the book *Programming Perl*, passwd(1) is rewritten in perl. The checks conducted are extensive: they contain case-folded dictionary checks, two-dictionary-word combinations[10], various suspicious patterns like telephone and social security numbers, license plates,[11] closely related ASCII sequences ("abcdef") and continuous keyboard sequences (e.g., "asdewq").

Retroactive

cops

cops is a system security checker that checks many more things than just password security. By default the only password security check is the check against the account name. The optional checks contain exact dictionary matches, GCOS checks and foraging for new dictionary words from some of the of the users' files containing personal information like .plan, .project and .signature. [Farmer92]

crack

crack is *the* retroactive cracker. It uses very fast encryption algorithms, intelligent ordering of guesses and complex configuration language which handles various mutations like

```
o → 0, i → 1, l → 1, s → $
```

yielding:

```
atlases → atla$e$
oratorial → 0ratorlal
```

and reversal, reflecting (drop → droppord), uppercasing and lowercasing. It can also use multiple machines simultaneously thus cracking the passwords in parallel [Muffett92].

ipasswd

Goals

The Computing Center HUT needed tools for enhancing password security. *crack* was the clear alternative as the weed killer. Each of the freely available proactive tools had some undesirable aspect: they were too vendor-dependent or their checks were inadequate or both. The mutation strategies used in *crack* would have been fine but they were built from the wrong, retrospective perspective. Besides, all the proactive tools tried to be plug-compatible replacements for passwd(1) or even for more like *shadow login suite*. They were not modular or portable. The Computing Center did not want a replacement for vendors' passwd(1) but rather a wrapper-like proactive judge to decide if new passwords were sufficiently resistant against retroactive cracking tools.

This need led to the idea of a server-client solution where the proactive server listened for client connections and gave its judgement on the password candidates. A problem arose: should the server's service point be local to each host or visible from the network? If the server were local, the cleartext password would stay local but the various databases, dictionaries, and executables would be duplicated at each host. If the service were unique and network-reachable, the dictionaries could be administered centrally but the password would travel in naked cleartext via TCP/IP. The network solution was chosen because the required authentication and encryption tools (Kerberos and DES) are available.

[10] Just two words concatenated is not enough: in between should be at least one non-alphabetic character.

[11] Patterns like these are strongly country-dependent.

Tools and Policies

A difficult separation task was evident: the *tools* and the *policies* must be separated. The server should be just a tool. It should only do the things the client requests and nothing more. The policy would be the password quality rules in each client. The client would also have to convey the information pertinent to each user at the moment of the password quality checking because the server does not intrinsically know anything about the users of its clients. The server may cache the policies and user information per client basis but not by default.

Another possibility is that only the disk-space consuming large dictionary checks would be served in the network and simpler checks would be done locally. As the prototype version was to implement only the dictionary checks, this did not make any difference.

Design Overview

Mutation Pipeline

The simplest of password quality criteria are:

- Minimum length
- Character set richness: enough different characters chosen from different character classes like alphabets, digits and punctuation
- Sufficiently different from the previous password in terms of changed characters
- Different from words closely related to the account: account name itself, GCOS field information, words from files like .signature

In the last two of these checks the client would have to supply the information to the server: for example the GCOS field and the previous password should be transferred to the server. These checks are quite simple and the main work was directed to the dictionary and mutation checks.

Various algorithms are available to make the required *fuzzy searching*: Levenstein edit distance which measures the distance between two strings, statistical methods which measure the probability that the given word 'a' is actually from the dictionary 'b', approximate grep program called agrep [Wu&Manber92] and precomputing the mutations beforehand. Because password checking is a time-critical problem, the precomputation scheme was chosen as the prototype server's method of finding mutations.

The database is generated by mutating an input stream from some standard dictionary file like /usr/dict/words. The mutations are stored in a standard dbm database for fast lookups:

```
mutate [options] </usr/dict/words |
  dbm [options]
```

There are many mutation methods that can be applied to a word:

- *casing:*[12]
  `UNIX → unix, Unix, ..., uNiX, ..., UNIX`
- *reversal*:
  `reverse → esrever`
  The reversal can be done at two different points: before other mutation methods as *prereversal* or after other mutation methods as *postreversal*. This difference may produce different outputs when combined with other mutation methods which depend on the ending of the word, for example the following method:
- Mutation that changes the stem of the word: a good example of this is irregular plurals like
  `mouse → mice`
  `vertex → vertices`
- All possible concatenated subselections of the subwords:
  `abc 123 def → abc, 123, def, abc123, abcdef, 123def, abc123def`
  This mutation method is called *partitioning*. It is normally used for checking various mutations of the real name of the account holder with the added twist that the initials-only versions should also be checked:
  `Jarkko Hietaniemi → jarkko, hietaniemi, jarkkohietaniemi, j, h, jhietaniemi, jarkkoh, jh`
  The lowercasing is possible because of footnote 12. The perturbed subselections where the last component, the last name, is swapped first should also be checked.
  `hietaniemijarkko, hietaniemij, hjarkko, hj`
- The *"real" mutation* where parts of words are mutated because of their visual or aural similarity to some other strings.
  - Substrings can look alike:
    `o → 0, i → 1, l → 1 s → $ s → 5, g → 9`
    `grisly → 9r151y`
    `positive → p0s1t1ve`
  - Substrings can sound alike:
    `too → 2, you -> u, for -> 4`
    `toobad → 2bad`
    `foryou → 4u`
  - These mutation types can of course be combined:
    `toofastforyou → 2fa$t4u`

[12] Because all mutations can be folded to lowercase and the word to be searched can also be folded to lowercase, only lowercase versions need to be stored. This is not the case with retroactive methods: every different casing must be separately tested.

The problem with mutations is that if a word has n potential mutation points, the number of mutations can be 2^n. It may be lower because mutations may overlap, for example in "you" there are two potential mutation points: "you" and "o". If the first point mutates to "u", the second one cannot mutate to "0" and vice versa. The first and the second mutations points are therefore *mutually exclusive*.

The number of mutations grows exponentially within each particular mutation method and when these are pipelined, these exponential terms must be multiplied and therefore the final size of output can be **very** large. The ordering of the mutation pipeline must be carefully thought out both to minimize the amount of data that flows forward and to avoid missing any potential mutations by destroying information. Building the pipeline is quite difficult because any portions of it can be skipped and therefore simple recursion is not enough. The pipeline in Figure 1 is used in the prototype.

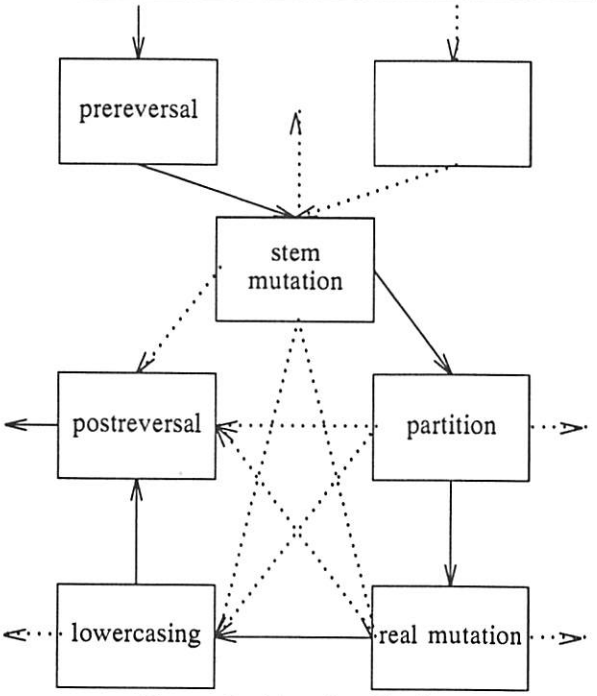


Figure 1: Mutation Pipeline

The pipeline is started from the top with prereversal or without it. The solid arrows follow the route of data during full mutation. The dotted arrows track alternate routes that can skip some portions of the pipeline. Some routes are not drawn to preserve clarity: those entering the pipeline from some point other than the two represented at the top are left out.

When the full mutation pipeline is applied to /usr/dict/words[13] the data multiplication factor ranges from 80 to 100 depending on the word. When duplicates are removed, the factor is approximately halved down to 50 thus expanding /usr/dict/words to a little more than 11 MB. The duplicates are caused by mutually exclusive mutation points. The factor depends strongly on the stem and "real" mutations used: in this case, 9 stem mutations and 13 "real" mutations were used. There is another limit that pushes the factor down: excessively short strings were not entered into the dbm database. Here the minimum length was three.

[13] 26880 words, 222518 bytes: a slightly amended AIX version

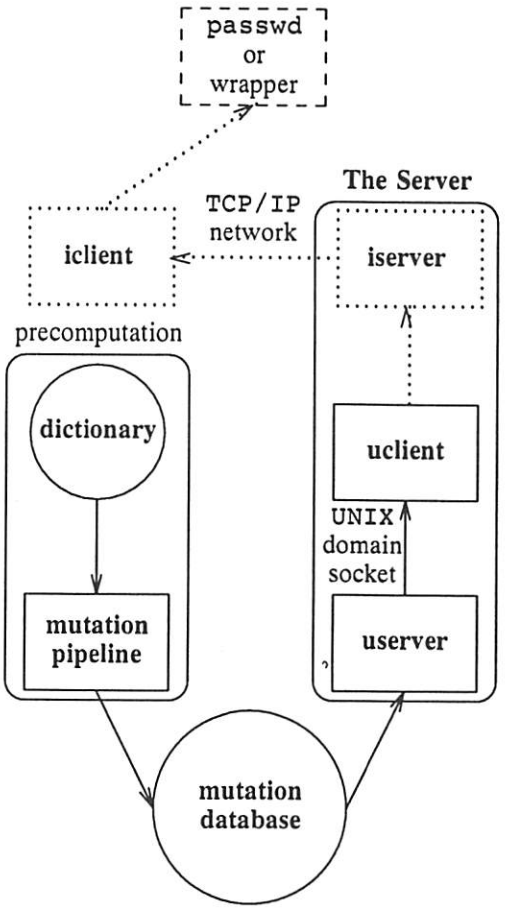


Figure 2: ipasswd

The Server

The server itself is layered: the lower part, **userver**, searches out the mutations directly from the precomputed **mutation database**. It is contacted by a UNIX domain socket and thus protected via the normal file protection scheme. The upper half of the server, **iserver**, talks to a client, **uclient** of the lower half of the server. The upper half of the server offers its service via an Internet socket to a client, **iclient**. **uclient** is the intermediary between the lower and upper halves of the server. Only those parts represented by solid lines were implemented to some degree in the prototype.

Protocol

The protocol implemented was very simple. In the following section "server" means the UNIX domain server which directly looks up potential mutations from the database and "client" means the UNIX domain client.

password After receiving this command from the client, the server waits for the client to enter the proposed new password. After issuing the new password the client waits until the server has stopped giving judgement about the password. After receiving the proposed password, the server gives its judgement. In this prototype version this meant listing all the possible basewords which the substrings of the proposed password could be mutations of.

quit After receiving this command, both the client and the server quit.

Below is a sample session with the server as seen by the client. For demonstration purposes *two* password candidates are tested at the same session, as opposed to the normal one candidate at a time.

```
password
c1t$0n9a
DICT/WORD:agnostic
DICT/WORD:gnostic
DICT/WORD:song
DICT/WORD:son
DICT/WORD:cit
DICT/COUNT: 3 3 3 4 4 4 3 1
password
t0day7h1gh
DICT/WORD:today
DICT/WORD:dot
DICT/WORD:ado
DICT/WORD:day
DICT/WORD:high
DICT/COUNT: 2 3 4 3 2 0 1 1 1 1
quit
```

The DICT/COUNT shows the cumulative hits on the proposed password, for example no hits were made on the "7" in the second case.

It is the duty of the client to study further the policy required and compare that with the evidence given by the server. The client gets a list of possible matches from the database and the policy it should obey from the Internet server. From these it should make its decision whether the password candidate is worthy to be a new password. This is not an easy task, here "*the policy*" begins. For example, are the two password candidates above "good enough"?

If the password quality checking were completely "serverized", it would also be the duty of the client to check the simpler rules like the minimum length. If on the other hand the simpler checks were localized, they would fall within the duty of the Internet client or even the passwd(1) itself or its wrapper. The duty of the Internet server would be the easiest: it will always be the simple service point between the Internet client and the UNIX domain client.

Performance

These performance figures are measured in an IBM RS6000m560 with 192 MB of real memory. The time measurements are always the sum of user and system times unless otherwise stated. Any real (wallclock) time measurement would have been meaningless because the system also had other processes than just *ipasswd*. The task was clearly separated into the precomputation phase and the server-client session phase.

Precomputation Phase

Mutation The mutations_generated/second is approximately 21000 while the input has uppercase letters as is in the beginning of /usr/dict/words, the first 22% of the file. When the lowercase words start, the speed is reduced to 17000/second, the average for the whole file being 18000/second. The mutations/second throughput stays reasonably constant except for the uppercase-lowercase difference.

Storing The dbm_store()/second starts from over 3000 and decays towards 2000. When the dbm file reaches a size of about 60 MB, the performance drops abruptly more than by half. Before the drop, average stores/second is 2400, after the drop it is only 1100, the overall average being 1800. The reason for this performance drop has not been examined further. The final size of the dbm file with full mutation pipeline is a little above 100 MB. The size verges on the impractical but on the other hand the lookups are very fast. One factor that enlarges the database is that it holds five kinds of data:

- mutations themselves
- baseword → baseword_index mappings. *basewords* are the original words of the input dictionary.
- baseword_index → baseword mappings.
- mutation → baseword_index mappings These last two mappings can be combined to reverse map mutations back to their respective basewords.
- metadata like the number of basewords and mutations

Again, one should remember that the size of the final output strongly depends upon the mutation rules used.

Because the generating of mutations was ten times faster than storing them, the performance of precomputation phase was strongly IO-bound, it used only 3-5% of the CPU capacity on the average. The

creation of full mutation database took from 8 to 14 hours realtime while the load ranged from 3 to 5 in the host.

Server-Client Session Phase

The UNIX domain server-client session was very fast: the communication went through a UNIX domain socket and the server made dbm lookups. The response time for an average length (6-10 characters) password was from 0.05 to 0.15 seconds.

Future Improvements

Algorithms

The large disk space consumption of the precomputed mutation database is a definite problem. To solve this alternative methods of mutation checking must be researched. For example *agrep* seems promising. It already has an option to search for up to 30000 matches simultaneously. The problem is that the matches must be exact.

Protocol

The userver-uclient protocol needs to be expanded to allow the client for example to

- Select which dictionaries to use in addition to /usr/dict/words
- Specify additional words to be checked; for example the account name and the GCOS information
- Change the parameters of the tests, for instance the minimum length

Portability

One portability problem is already at least partially solved, namely the byteorder dependency of the dbm files. This may be a problem when transferring mutation databases between architectures. When creating or opening the database for reading and writing the dbm routines automatically append a string like "_4321" or "_1234" to the name of the database, representing the byteorder of the database host. Using this scheme, the routines rather fail than open wrong databases. This is not a "pretty" solution, a cleaner solution would be to write a custom-made byteorder-independent database system. Many more portability problems certainly lurk around the corner.

Authentication and Encryption

Before any network service can be supplied, authentication and encryption schemes must be developed. These will be most probably based on Kerberos and DES.

Availability

ipasswd exists only as a very crude prototype implementation. Foolhardy volunteers may receive beta versions in the summer of 1993 by contacting Jarkko Hietaniemi, <Jarkko.Hietaniemi@hut.fi>.

Conclusions

Password security can fail due to many reasons. The most important thing that can be done is to organize comprehensive user education. Only when the users *know* why and how to choose good passwords, can they be blamed for using bad passwords. To prevent and to weed out poor passwords the administration needs efficient tools. The *retroactive* methods are very CPU-intensive. Their smooth organization in a heterogeneous UNIX network is difficult. The *proactive* methods are very fast, although disk-consuming. The proactive checking is conducted always exactly when the checking is needed: when the password is changed.

Acknowledgements

My thanks to Jukka Virtanen and Tero Mononen for technical assistance and proofreading, to David Brearley, Paul Bibire and Jouni Malinen for proofreading and to Timo Sirkiä for making me start writing this paper in the first place.

Bibliography

[Bishop92] Matt Bishop, *passwd+* alpha version 4 program documentation.

[Brand90] Russell L. Brand, *Coping with the Threat of Computer Security Incidents – A Primer from Prevention through Recovery*, available from CERT.

[Farmer92] Dan Farmer, *cops* v1.04 program documentation.

[Garfinkel&Spafford91] Simson Garfinkel and Gene Spafford, *Practical UNIX Security*, O'Reilly & Associates, ISBN 0-937175-72-2.

[Grampp&Morris84] Frederick Grampp and Robert H. Morris, *UNIX Operating System Security*, *AT&T Bell Laboratories Technical Journal*, vol 63, no 8, October 1984, pp. 1649-1672.

[Haugh92] John F. Haugh II, *shadow* v3.2.1 program documentation.

[Hoover92] Clyde Hoover, *npasswd* v1.2 program documentation.

[Klein90] Daniel Klein, *"Foiling the Cracker": A Survey of, and Improvements to, Password Security*, Software Engineering Institute, Carnegie Mellon University.

[Morris&Thompson79] Robert Morris and Ken Thompson, *Password Security: A Case History*, Communications of the ACM, vol 22, no 11, pp. 594-597, November 1979. Also in the *UNIX System 7 Programmer's Manual: the Supplementary Documents* or the *System Manager's Manual.*

[Muffett92] Alec Muffett, *crack* v4.1 program documentation.

[Ritchie79] Dennis Ritchie, *On the Security of* UNIX, UNIX System 7 *Programmer's Manual:*

the Supplementary Documents or the *System Manager's Manual.*

[Spafford88] Gene Spafford, *The Internet Worm Program: An Analysis*, Purdue Technical Report CSD-TR-823, November 29, 1988.

[Stoll89] Clifford Stoll, *Cuckoo's Egg*, Doubleday, 1988, ISBN 0-671-72688-9.

[Wall&Schwartz91] Larry Wall and Merlyn Schwartz, *Programming perl*, O'Reilly & Associates, ISBN 0-937175-64-1.

[Wu&Manber92] Sun Wu and Udi Manber, *Fast Text Searching With Errors*, University of Arizona Tucson Technical Report TR 91-11 and *agrep* v2.04 program documentation.

Author Information

Jarkko Hietaniemi works as a systems analyst in the Systems Support Division of the Computing Center, Helsinki University of Technology, Finland. He is also struggling towards a MSc in Computer Science in the same university, that is, when he is not questioning poor innocent passwords with white-hot tongs and knotted whips. You can reach him by electronic mail at Jarkko.Hietaniemi@hut.fi.

DeeJay – The Dump Jockey: A Heterogeneous Network Backup System

Melissa Metz & Howie Kaye – Columbia University Academic Information Systems

ABSTRACT

Prior to 1991, we ran our systems' backups using manually loaded 9-track tapes. Due to increases in disk capacity and our desire for unattended operation, we decided to purchase an 8mm tape jukebox. Using this, we hoped to run automated backups of as much data as possible across an assortment of platforms — including Suns, Encores, NeXTs, Macintoshes and Novell servers.

After evaluating several commercial backup products, we decided to write our own. DeeJay consists of a backend server and several client programs which communicate across our campus network. We designed it to accept data from any program which can write to standard output, so it is not limited to *dump*(8) and *tar*(1). In order to maximize tape usage, we use the concept of virtual tapes of "infinite" length. DeeJay writes tapes which conform to the ANSI standard tape format, bundling multiple savesets together on a virtual tape which can span many physical tapes.

The backend, **dj**, acts as the virtual tape interface, transferring data between the network and the physical tape device. It also maintains a database of tape and jukebox contents, as well as information concerning which drives and tapes are in use. In order to run fast enough to finish the backups of our growing number of workstations overnight, dj uses asynchronous I/O and double-buffering.

The backup coordinator, **coback**, mounts a virtual tape for full or incremental backups and then directs each host in turn to run the backups that belong on that tape. Each host runs **djback**, which reads a backup schedule and runs the appropriate set of backups.

To allow for manual tape drives, as well as different types of automatic tape mounters, each tape drive has mount and unmount utilities associated with it. These utilities send the low-level loading commands to the device (human or otherwise). Thus, DeeJay can support many different types of tape loaders simply by plugging in new mount and unmount utilities.

Site Description

Like most university computer centers, Columbia's Academic Information Systems (**AcIS**) has many different types of machines networked together, all requiring backups, and no desire for staff members to mount tapes all day. Limiting the list to our UNIX systems, we have an Encore Multimax, two Sun 4/490s, two Sun 4/280s, eleven (and counting) SparcServers (headless SparcStations), dozens of NeXT workstations and NeXT cubes, assorted Sun workstations, and a history of Ultrix machines. In addition, we have a few Novell fileservers, which we are responsible for backing up. There are also many Macintoshes and PCs that are rarely, if ever, backed up.

History

Back in the old days, our computers — we had DECSYSTEM-20's — came with tape management systems. They wrote DUMPER format tapes, and verified internal tape labels before writing to a tape. Backups were done to 9-track tapes, which were loaded by operators all night long.

When we got our first UNIX systems, we were surprised by how primitive the backup "system" was. We wrote ourselves some scripts that went through a hardcoded list of filesystems, asking the operators which to back up, and prompting for the next tape to be mounted. While the tapes had external labels stuck on (eventually even *color-coded* ones), they did not have internal tape labels, and didn't for another five years. We held our breath for those five years, hoping the right tapes got loaded each night.

We went through a series of backup schemes. One read the *dumpdates*(5) file to see which backups needed doing. Another was a C program based on the original scripts, made prettier by use of a **CCMD**[1] command interface. We got as far as removing the filesystem- and host-specific information from the program and putting it into configuration files. Still, our backup systems

[1] CCMD is a friendly command interface including command completion and built-in help. It was written at Columbia by Andy Lowry and Howie Kaye.

continued to go through a list of filesystems, requesting tape mounts from the operators and writing savesets to local tape drives. Each filesystem was written to its own tape (or tapes).

When we got our two Encore Multimaxen, we decided not to pay the extra money for even more 9-track tape drives, and extra cabinets to hold them. The Encores came, by default, with drives that used weird square tape cartridges, which we could only purchase direct from the manufacturer. Our operations staff was unhappy with yet another (non-standard) type of media, and in addition the drives were annoyingly slow at loading tapes. This reinforced our intention to run remote backups. Soon, we had integrated *rdump*(8) into our backup program, to back up the Encores to the tape drives on our three Sun 4/280's. The tapes were still being mounted by operators, who could not spend all of their time watching the UNIX systems and waiting for tape requests.

By 1991, several of our staff workstations were being backed up to 8mm tapes during the day by staff members who had other primary responsibilities. With our growing disk capacity and desire for a more "lights out" operation, we decided to purchase an 8mm jukebox. Using this, we hoped to run unattended backups of as much data as possible, across an assortment of platforms.

We purchased a **Summus Jukebox**, which holds 54 8mm tapes in a round carousel,[2] and two tape drives. Eventually, we moved on to a 116-tape **Exabyte EXB-120**, (called an *IceBox* by our reseller) with four of the new 5-gigabyte tape drives.

Rejected Alternatives

In order to run our backups on the jukebox, we wanted a system that could send the commands to the jukebox to mount the right tapes, and a system that could deal with all the different machines we wanted to back up. Before writing our own system, we looked at several alternatives.

rdump, as used in our pre-jukebox systems, was unacceptable for several reasons. The main one was the basic *dump*(8) philosophy, which allowed for one saveset being written to a number of tapes, but not for a number of savesets being grouped together on one or many tapes. Dump keeps track of how much data it writes, and given the length and density of a tape it calculates when it expects to reach the end of the tape. If the tape is short (as they sometimes are), or if dump doesn't start at the beginning of the tape (as when we put multiple savesets per tape), it can get an end-of-tape error. When this happens dump is unable to recover, and aborts the backup.

Another problem was that rdump didn't work everywhere — the version of rdump running on our Ultrix machines used a slightly non-standard protocol, incompatible with the *rmt*(8c) running on our Suns.[3] We also had problems with our NeXT machines. In the *dump*(8) man page on the NeXTs, it explicitly states that "Because of the interworkings of rmt and rdump, it is only possible to run rdump from one NeXT machine to another."

Apart from the way dump handled tapes we liked the way it worked. It's the standard BSD backup program, and unlike other backup programs it handles every file[4] on the filesystem — including special devices and odd filenames. Dump could be told to write to standard output instead of a tape, leaving us free to do our own tape handling.

Of course, we had to consider various commercial packages as well. We got demo versions of a few, and tried them out.

One of the most well-known systems (certainly according to the number of articles we've seen about it) is **Legato Networker**. Unfortunately, Networker uses their own dump format, which means it is only available on systems they have ported it to. At the time, this included Suns and perhaps DOS machines. They had no plans to port it to the (somewhat obscure) Encore machines. According to a recent article in *Network Computing*, they currently have clients for "most brands of Unix workstations as well as DOS and Macintosh" and have recently announced a version for Novell NetWare.[5] However, we have great faith that companies can come out with new workstations and new versions of operating systems faster than any software vendor can come out with new versions of their software.

When we first purchased our Jukebox, the **BudTool** system did not have a driver for it, though they did have a driver for the EXB-120 (which we didn't have then). We tried out BudTool's point-and-click interface, but prefer a nice flat file we can put into *RCS*, since revision control of system configuration is almost as important as backups.

Long before we moved to 8mm tapes, we had tried a couple of other backup systems. One of them

[2]Given a standard 19-inch rack, the size of the 8mm cassettes, and simple geometry, a circular carousel can contain just 54 tapes. Thus, other brands also contain the magic number 54.

[3]We heard rumors later that there was a switch to the Ultrix rdump to make it work with the other rmt protocol, but never found the time to get it working.

[4]For exceptions, see Elizabeth D. Zwicky, *Torture-testing Backup and Archive Programs: Things You Ought to Know But Probably Would Rather Not*, LISA V Conference Proceedings, San Diego, CA, September 30 - October 3, 1991, p. 181-190.

[5]Art Wittmann, Todd Tannenbaum and Jim Drews, "Industrial-Strength Backup: Legato Networker for Unix and NetWare," *Network Computing*, July, 1992, p. 52.

dumped core a lot, and the other demanded that an operator be constantly logged in somewhere. One allowed writable tapes to be mounted when *read-only* was clearly specified. These experiences made us wary of commercial backup software, and reinforced our decision to write our own system.

Design

DeeJay was designed with certain key features in mind. First of all, it was envisioned as a network server which would allow hosts across our network to access a set of tape drives. It was expected that these tape drives would have some sort of automated tape mounting scheme (a jukebox of some sort), but we felt it was important to allow manually mounted drives to fit into our scheme.

Second, the software had to allow a large variety of machines to connect to it to do their backups. Like most universities, we have a heterogeneous network made up of many different types of systems. We felt it was important to be able to support many of these systems, and to use the native backup software on each.

Third, we wanted to use the tapes efficiently. Given that we were going to be using 8mm tapes which could hold up to 2.3 (or even 5) gigabytes of data, we wanted to allow multiple savesets on each tape. Otherwise, we would be wasting a lot of tape and changing the tapes in the jukebox too often. To make sure we filled tapes to the end, we wanted to write as many savesets per tape as possible. Rather than worrying about the actual length of the tape, and the possibly varying amount of data that would be written to it each week, we wanted to continue seamlessly on to the next tape after filling a tape.

We used the concept of **virtual tapes** or **tapesets** of "infinite" length. Dump does the same thing, moving on to the next tape in a tapeset when it expects the current tape to fill. However, we had dealt with the problems of dump's algorithm, and wanted a system which would handle end of tape conditions whenever they occur, and not just when they are expected.

In our design the server would allocate a new tape whenever it needed one (assuming there were unused tapes available), performing whatever bookkeeping was necessary along the way. Via the network interface, the client process would see only the infinite-length tapeset, and need never know that it might span several tapes. In this way, the actual task of handling end-of-tape (EOT), switching to another tape (if possible), and continuing the backup could be concentrated in a single place.

Fourth, we wanted labeled tapes. Especially given an automated system, we wanted to make sure that our backups didn't get overwritten. Since a standard already exists for writing multivolume labeled tapes, we decided to use it.[6]

Any backup system is useless unless restorals can also be done. While we don't do many restorals, we needed the restoral procedure to be simple — at least as easy as it had been when we were using 9-track tapes. We had to be able to look up the location of a particular saveset easily, and position the correct tape to that location without too much trouble. We also needed to get the data on that tape to the right restoral program, since it was conceivable that no software on our tape-host would be able to read some other host's backups. Some mechanism for maintaining and searching the table of contents of a dump would also be needed.

Since we had been doing backups all along, we kept our backup scheduling system. We do daily incrementals of most filesystems, and weekly fulls.[7] We keep each incremental for two weeks before overwriting it, and each full for three to four weeks. The first fulls done each month are designated as *monthlies* and are kept for three months.

To allow for a more rational tape-reuse policy, we wanted all the savesets on a given virtual tape to have the same expiration date, since overwriting any data on an 8mm tape makes it all inaccessible. So, each tapeset would contain either incremental backups for a certain day or fulls for a certain week, from multiple hosts and filesystems.

For the hosts to communicate with the tape server, we designed a simple *net-ascii* protocol which allowed us to perform all of the basic tape operations, as well as a few bookkeeping operations to keep track of what tapes we had and what was on them.

On each host to be backed up, we wanted a program that would establish a connection to the tape daemon, mount the correct tape, and then send the backup data across this network connection. It would not be aware that as each physical tape filled up a new one was mounted.

Rather than rewriting dump, this program would simply run dump with its output redirected to the network connection. Any errors from the tape daemon would be handled by this backup program. If a program other than dump was needed, it could simply be plugged into this scheme. Any program which writes data to standard output could be used.

---

[6] American National Standard Magnetic Tape Labels and File Structure for Information Exchange (X3.27—1978).

[7] The scheduling also allows for filesystems which rarely change, like vanilla sources, which are given full backups monthly and no incrementals. Also, the partition where netnews is kept changes too often to bother backing up, and is easily recoverable by just waiting for more news, so we only run monthly fulls on it, to keep track of the basic structure.

Mounting an 8mm tape and winding it past existing savesets, takes a long time and tends to wear out the tape. To minimize extra mount and unmount commands, and to keep backups orderly, we needed a backup coordinator. The coordinator mounts a tape and then directs each host that might use that tape to do the appropriate backups. This also makes sure that only one host at a time tries to access the tape, and that each host starts up as soon as the previous one is done.

With the multiple tape drives, we can run simultaneous backups, one on each drive. However, we can't use two tape drives if two hosts are trying to write to the same tapeset, so we divided our hosts up into groups. This also allows us to schedule them differently — lightly used timesharing machines in the evening when we can keep an eye on them and get an early warning of upcoming failures, larger systems a little later, and staff workstations in the early morning when everyone finally goes home. Each of these groups is run by a separate instantiation of the coordinator.

The DeeJay Protocol

The DeeJay protocol was designed along the lines of **SMTP**[8] (the Simple Mail Transport Protocol) and is also similar to **FTP**[9] (the File Transfer Protocol). A control port is used to manipulate a tape (and drive), and as in *ftp*(1c) a secondary data port is used to actually send the data to be written to tape. Each command is issued as a single line of text. Error codes are returned with optional comments following them. Multi-line responses are given by following the error code with a "-". The final line of a multi-line response has no "-". The commands in the protocol are listed below.

MOUNT [-e expiration-date] [-s] TapeSetName
Associates a tape drive with a tapeset. If there are no tapes yet in the tapeset, one will be allocated at this time and labeled. Otherwise, a tape is not physically mounted until some positioning command is given.

UNMOUNT
Unmounts the currently attached tape, returns it to its slot in the jukebox, and frees the tape drive.

ATTACH TapeSetName
Attaches to an idle tapeset. Attach and detach allow a tapeset to be used by multiple clients without being repeatedly mounted and unmounted.

DETACH
Detaches from the currently attached tapeset. This leaves the tapeset mounted, and attachable by some other process. If a client closes its control connection to the server, it does an implicit detach.

REWIND
Rewinds the currently attached virtual tape to the beginning. Since a tapeset can contain many physical tapes, this may involve unmounting the current tape and remounting the first tape of the set. It is not used during normal nightly backups.

WIND
Forward spaces the current virtual tape past the last file. Since we keep track of what is on each tape, this can be done efficiently by mounting the last tape in the tapeset and forward spacing the appropriate number of files on it. A wind at the end of a virtual tape does nothing. A wind command must be issued before the tapeset can be written to; that is, you can only append to a tapeset.[10]

FSF n
Forward spaces the tapeset by *n* files, mounting and unmounting tapes as needed. As with the wind command, this can be efficiently done by mounting the right tape and moving to the correct spot on it.

BSF n
Backward spaces the tapeset by *n* files, as in the FSF command.

GOTO m n
The absolute addressing method of tape movement. This positions the tape to the *n*th file of the *m*th physical tape. This is most useful when a read operation is about to be done (for restorals).

WRITE port hostname filename level "info" [backup-hostname]
Tells the server to connect back to some **port** on the specified **hostname** to receive data. The **filename**, **level**, **info**, and **backup-hostname** are used in the tape labels and are saved in the database of savesets. Once this data connection is established, the server will write whatever data is received to tape, allocating new tapes as

[8]Jonathan B. Postel, "Simple Mail Transfer Protocol", Internet RFC 821, 1982.

[9]Bhushan, Abhay, et al, "File Transfer Protocol", RFCs 114, 172, 354, ..., and 959, 1971-1985.

[10]As it says in the *mtio*(4) man page, "Writing is allowed [...] at either the beginning of tape or after the last written file on the tape." However, DeeJay does not allow writing at the beginning of a tape, unless the tape is empty.

necessary. Each tape and saveset is properly labeled by the server in the ANSI format. An end-of-file on the data connection signals the end of the saveset. At this point, any error codes are returned on the control port. The **DATA** command is a synonym for **WRITE**.

READ port hostname
As with the write command, a connection is made back to the specified **hostname/port** pair. The current file is then read from the tape and written over the network. If the network connection is closed before the data is all read, the tape is moved to the next file. When the entire file has been sent, a status code is returned. The **RDATA** command is a synonym for **READ**.

STATUS
The status command returns various information about the currently attached saveset and drive.

OUTDATE tapeset
If the expiration date for the tapeset has passed, all savesets on the tape are marked as expired, the tapes that were in that tapeset are marked as not in use, and the tape-count for the tapeset is set to zero. (If the expiration date is still in the future, nothing happens.) This will permit the tapes to be reused. However, until each tape is actually reused, the records for the savesets on those tapes remain in the database, since those savesets are still recoverable. A **WIND** command on an expired tapeset will leave it at the beginning, since the tape-count is zero. The expiration date for a tapeset is set the first time it is mounted (or the first time it is mounted after having been expired).

ADDFILES port hostname saveset
Adds a list of files to the table of contents for a saveset. As with **READ** and **WRITE**, a separate data connection is used to transfer the list of files.

DIR hostname filename starttime endtime
The DIR command searches the table of contents files of a given host for files matching the specified filename written between the start and end times. This is useful for searching the contents of various dumps, to determine the best tape on which to find a given file or directory. The specified filename may contain wildcards.

QUIT
Ends a session.

HELP [command]
Lists all the available commands, or gives brief help on the specified **command**.

Here are some sample DeeJay sessions.

```
Client          Server
MOUNT W1MT
                000 OK (/dev/nrst12)
WIND
                000 OK
WRITE 5000 watsun WGW1MT 0 "/usr"
                000 OK
WRITE 5000 watsun WKW1MT 0 "/pub/kermit"
                000 OK
UNMOUNT
                000 OK
```

Above, we see a client mount a tapeset, wind it to the end (where it can write to it), write two savesets, and then unmount the tape. Note that the name of the physical drive is returned, though the client never uses it. (Sometimes the humans do.)

```
MOUNT M3WK          000 OK (/dev/nrst13)
GOTO 2   5          000 OK
READ 5000 cunixd    000 OK
UNMOUNT             000 OK
```

In this example, a client mounts a tape, positions it, and reads the file at that position. The positioning information might be obtained earlier using the **DIR** command to search for a file.

The DeeJay Database

The DeeJay database contains several tables and a set of lookaside files. Our implementation uses INGRES, a relational database system. All of the database code is written in **EQC** — C, with embedded QUEL (Query Language) constructs.[11]

The **carousel** table lists which tapes are loaded in the jukebox (our original jukebox had a removable carousel which held the tapes, hence the name). The carousel table associates each slot in the jukebox with an external tape label.

The **drive** table contains various information about each tape drive. Some of this is state information, like whether dj needs to write a tape mark before writing data, what tape set is associated with the drive, what tape is currently mounted (and which slot it came from), or what the current tape position is. This information must be maintained in a shared database, since one dj process may mount and position the tape while another may write to it.

The drive table also specifies properties of the drive, like which driver dj will use to handle automatic mounts and unmounts, and the optimal blocksize for writing. Finally, it contains statistical information on tape usage and error rates. These statistics are useful for determining if a drive needs

[11] A site without INGRES could rewrite the database module (all database code is localized in one file) to use an alternate DBMS.

cleaning or service, or if some connection has timed out and an in-use drive should be freed up.

The **tape** table contains information for each tape (whether or not it is currently in the carousel). It maps external labels (those written on the outside of the physical tape) to internal labels. The tape table also contains statistical information about each tape — the date it was put in service, the last write date, the error rates, and the write count. This is useful for determining when a tape should be replaced. Dj mainly refers to tapes by their internal labels, which are usually based on the name of the series that the tape currently belongs to. The external label on a tape uniquely identifies it, while the internal label can change when a tape gets reused. Similarly, there may be several tapes called "BLANK", but each one will have a different external label and different usage statistics. The tape drive we currently have has a bar code reader on it, so our external labels are sequentially numbered barcodes. When mounting a tape, dj looks up the external label for the tape that it wants, and mounts that tape, verifying that the bar code is the one it expects in that slot. Dj then reads the tape's header to verify that the internal label matches the one it expects.

The **series** table describes a series, or tapeset. It holds the expiration date for the tapeset and the count of physical tapes currently allocated to the tapeset. If the tapeset has been expired (via the outdate command), the tapecount is set to zero. The tapes in each series have internal labels consisting of the tapeset name followed by a digit (count).

The **saveset** table maintains the list of savesets in each series. It also keeps track of the starting and ending tape positions for each saveset. There are several ways of looking up a saveset. It has an explicit name, which is used in the file header label on the tape. It also has some backup-specific information like the filesystem name, hostname, and the date and level of the backup. The saveset table is primarily used during restorals while tables like **drive** and **series** are used while backups are being done.

Tape Labels

With our pre-existing scheduling system, we had a naming scheme based on one saveset per tape. The name incorporated a letter indicating the host name, a letter for the filesystem, two letters indicating daily (and which day of the week), weekly, or monthly, and a number to indicate which set of dailies, weeklies, or monthlies. For example, a weekly backup of cunixF's root partition would be called **FA1WK**. We kept these names for the individual saveset labels.

To name the tapesets, which each contain backups of several hosts and filesystems, we simply drop the first two letters of the saveset names (indicating host and filesystem) and keep just the type and number (e.g., **1WK**). Each tape is named by adding a sequence number to the tapeset name — **1WK1**. Finally, we added an extra letter (to all the names) for groups of hosts that were done separately (**G1WK**).

Tape 1	
	Tape 1 VOL1
File 1	File 1 HDR1 File 1 HDR2 TAPEMARK File 1 Data TAPEMARK File 1 EOF1 File 1 EOF2 TAPEMARK
File 2(a)	File 2 HDR1 File 2 HDR2 TAPEMARK File 2 Data TAPEMARK
	Tape 1 EOV TAPEMARK
	TAPEMARK

Tape 2	
	Tape 2 VOL1
File 2(b)	File 2 HDR1 File 2 HDR2 TAPEMARK File 2 Data TAPEMARK File 2 EOF1 File 2 EOF2 TAPEMARK
	TAPEMARK

Figure 1: ANSI Tape Labels

These tape and saveset names are used in the **ANSI** labels. Following the standard, we place a volume header at the beginning of every tape, which contains the tape name, a date, and various other information.

Each saveset (except possibly the last on the tape) is made up of three tape files. The file header consists of two blocks, called **HDR1** and **HDR2**, which contain (among other things) the saveset name. The next file is the saveset data itself, followed by the file trailers (or end-of-file label), also two blocks.

If a saveset spans two tapes, it has an end-of-volume label at the end of the first tape, a new header label on the second tape, the continuation of the saveset, and then finally an end-of-file label at the end (see Figure 1).

Each tape ends with two tape marks.

Server Implementation

The server, **dj**, is split into several layers. There is a *yacc* (1) based parser which accepts commands from a network connection (or from standard input when various debugging switches are supplied). This implements the DeeJay protocol, and is really the driver for the program. When commands come in, the parser calls the appropriate high-level dj functions. These are responsible for passing the data to the database and/or tape-handling layers. Finally, there are several drivers, one for each supported jukebox type (currently there are two), which interact with the specific automounting hardware on that jukebox.

In order to read and write tapes as quickly and efficiently as possible, dj implements a double buffering scheme using asynchronous I/O. Data is read from the network and then written asynchronously to tape while more data is read from the network, filling the next buffer. When the write actually completes, dj is notified via a **SIGIO** signal, at which time it will do the next write. Dj uses SunOS's *aiowrite* (3) system call to do this.[12]

DeeJay was designed to handle different types of tape carousels by specifying different mount and unmount commands in the drive table. These commands can then contain the specific information for their own particular brand of jukebox. However, the mount and unmount commands we wrote (**djmount** and **djunmount**) can send commands to either an Exabyte EXB-120 or a Summus Jukebox, using a table indexed by drive type to choose the lowest level commands. The drive type is saved in the drive table of the database.

DeeJay also has the ability to clean the tape drives periodically. There is a "clean interval" in the drive table, which governs this behavior.[13] Cleaning is implemented through the **djclean** program, which just mounts the cleaning tape, waits a while, and then unmounts it. It looks for tapes with the name "CLEAN*". Djclean keeps track of how many times each cleaning tape has been used (our cleaning tapes can only be used twelve times), and sends mail when a cleaning tape needs changing.

To make restorals easier, we added a mechanism for keeping track of the contents of a saveset. Our original implementation attempted to keep a listing of filenames in the database with our other data. This turned out to be a poor idea. Variable-length filenames do not map well to fixed-length database fields. Instead, the file lists are maintained in separate files, named by their saveset names. The files are kept compressed[14] since they can get large. The lists are searched with a wildcard matcher similar to that of *csh* (1.)

When data is being written to tape, more tapes are allocated automatically as necessary. If an expired tape with the correct label exists, it is used. Otherwise, the tape and carousel tables are searched for any blank tapes (those with the name "BLANK") which are currently in the carousel. If there are any, the system attempts to label one of them as the next tape in the current tape set. If the relabel fails, DeeJay goes on to the next blank tape. If no blank tapes are available, it next looks for any tapes which are labeled, but are expired. If it finds one, it attempts to relabel it. If no expired tapes can be found, an end-of-tape error is returned to the writer.

Eventually, we surpassed the capacity of our 54-tape jukebox, and had to reload it periodically. So at any given time, some tapes were on the shelf and not in the carousel. In order to make sure that DeeJay wasn't trying to use any of the tapes that weren't in the carousel, we wrote the **playlist** program. Playlist predicts which tape series will be used in the next several days, and checks whether all the tapes in those series are in the carousel. If not, it lists the tapes that need to be loaded. Playlist is generally run by cron, and its output is mailed to the backup user (which should be forwarded to the administrators in charge of DeeJay).

[12] Currently, this means that dj can only run on a Sun machine, under SunOS 4.1 and greater, since Sun's *aio* system calls were introduced in release 4.1 of the Operating System. They rely on an implementation of lightweight threads within the kernel to allow multiple execution paths in a single process.

[13] Cleaning the drives periodically was in our initial plans, but became more urgent when we started getting a large number of tape errors due to dirty drives. These were caused by paper dust in the room with the tape drives and by the lack of filters on the Summus Jukebox.

[14] We use the *compress* (1) family of programs which implement the Lempel-Ziv compression algorithm.

Client Implementation

The backup coordinator, **coback**, is run out of *cron*(8) each night, with a list of hosts to back up and what level of backups to do — fulls, incrementals, or both.[15] Coback determines which tape should be used today (or this week) for this level of backup, and requests that dj mount the tape. It then detaches from the tape, and tells each host (or **backee**) in turn to run the backups which belong on that tape (see figure 2). When all the hosts are done, it attaches to the tape again and unmounts it. Coback usually runs on the same host as the dj daemon itself, just to have fewer points of failure (only the one host), but this is not necessary.

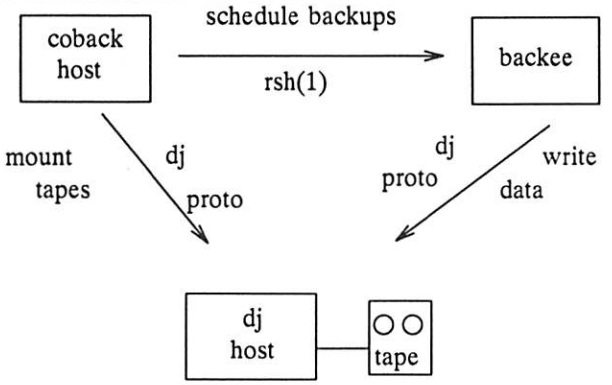


Figure 2: DeeJay System Overview

Several coback processes can be running at one time, each associated with a different group of hosts and a different set of tapes, and each using a different tape drive.

djback, running on each host, reads configuration information from the **xfstab** (extended **fstab**) file, such as which filesystems need backups and how often. It connects to dj, and issues an ATTACH, a WIND, and finally a series of WRITE commands, one for each scheduled backup. For each backup, it runs *dump*(8), with its output redirected to the data port from the WRITE command (see figure 2). When all the backups are done, it detaches from the tapeset.

Each filesystem has its full backups done on a certain day of the week, specified in the xfstab. If coback instructs djback to do a full backup, and no full backups are scheduled for this day of the week, djback exits without even attaching to the tape. This is especially useful if no backees have backups for today — no WIND command is ever issued, and thus the tape need never be mounted, saving much wear and tear as well as time.

[15]Our daily backups consist of incrementals of all filesystems followed by the fulls scheduled for today. We run incrementals on even those filesystems about to have full backups, in order to have two ways of getting to that data — in case we have any problem reading the tape.

Coback uses *rsh*(1c) to run djback on the backees. In order to detect failures, a shell wrapper is placed around the djback command, so that the backee prints the exit value from djback, flagged so that coback can recognize it.[16] So, if djback runs into trouble, coback can tell what kind of error occurred — a host error, like "permission denied" trying to run dump; or a dj error, like not being able to access the tape. This way, if there is a problem such as the tape becoming unusable when a backee tries to write to it, the rest of the backees will not make futile attempts to do the same thing. By exiting as soon as a tape error occurs, coback can keep track of where the backup failed, leaving the information in a checkpoint file.

The checkpoint file contains, first, the *pid* (process ID) of the coback process. This way, another coback process trying to pick up from that checkpoint can tell that it is still being written to. Coback then lists the labels of the full and incremental tapes it is planning to use, and the day of the week it is working on. Finally, it indicates which level of backups it is doing (full or incremental) and which host it last started.

Djback also writes checkpoint files. Like coback, it includes its pid. Next, it lists which filesystem it is backing up, at which level.

If dj or the tape-host crashes in the middle of a backup, djback gets an error when it next tries to write to the socket, and it aborts. It leaves the checkpoint file, indicating the filesystem it failed to backup. Coback recognizes that djback failed on that backee due to a tape error, and aborts the entire set of backups. Later, once the failure has been cleared, coback can be run again in checkpoint mode, and it will pick up with the backee which failed. djback will then start with the filesystem it was writing at the time of the failure, and continues with the rest of the filesystems.

Normally, djback uses *dump*(8) to back up each filesystem. However, hosts can be configured to use an alternate backup program[17] — for example, we might use a program like *cpio*(1) or *tar*(1) if we thought a certain host had quickly-changing filesystems that were more susceptible to the problems dump has with filesystems changing under it.

In order to send a listing of the files in a dump to the server, the output from dump is duplicated[18] and passed to the *restore*(8) program, run with

[16]Otherwise, the return value from *rsh*(1c) merely indicates whether the host was reached.

[17]Currently, alternate backup programs are specified on a per-host basis, but in the future we will probably expand this to allow per-filesystem backup programs.

[18]We wanted something like *tee*(1) for this, and wrote **dumptee** to quickly output to a pipe and *stdout*, ignoring the pipe once the subprocess (restore) exited.

switches to have it list the table of contents of the dump. The assumption here is that a given host is most likely to have a restore program which will understand the format of that system's dumps.[19] The output of restore is filtered to get just a list of the files, and then piped into the **dumpdir** program, which sends it to the server using the **addfiles** command. When using *cpio*(1) for backups, a list of files to back up is generated first. This list can be used directly with dumpdir to update the table of contents in the server.

Coback and djback handle certain special cases for hosts that are unable to do their own backups. For example, we back up Novell fileservers by mounting their filesystems via Novell's NetWare NFS on a UNIX host, which is known as the **helper**. Instead of connecting to the backee (which may not even be possible), coback connects to the helper host, sending an option to djback specifying the actual backee. Generally, this backee would have an alternate dump program specified — in our case, we use a script that runs a *find*(1) and then *cpio*(1) with the output of the find. The script implements incremental backups with a "dumpdates" directory containing files analogous to the lines in a normal dumpdates file, which are used in the find command.

We hope to implement Macintosh backups in a similar fashion, perhaps using the University of Utah's *Macintosh to UNIX Dump System*, which already seems to have the concept of a helper host which contacts the backee to ask for the data.[20]

Since djback, at its lowest level, runs a program with its output redirected to the DeeJay server, a pipe interface seemed natural. So, the **djpipe** program was written. It will either read from *stdin* or run a program specified on its command line.[21] Either way, the output of the program gets passed to the server and written to tape. Djpipe requires a tapeset and tape-host on the command line so that it can attach to the correct tape, which is assumed to be already mounted. djpipe is basically djback without the scheduling, and requires less configuration.

The corresponding tape-reading program, **djcat**, was also written. Djcat also requires a tapeset and tape-host. Like djpipe, djcat assumes that the tape has already been mounted and positioned. Djcat attaches to the host, and issues DeeJay commands to read the current file. As in djpipe, either the output is sent to *stdout*, or a specified program will be run with the contents of the tape as its standard input.

[19]For example, a dump written by one of our NeXT systems can only be read by NeXT's version of restore.

[20]The leaflet I have doesn't seem to mention a name for this package, which is cheap (maybe $250/site) but not free. brian@cs.utah.edu is listed as the technical contact.

[21]Having djpipe run the program with its output redirected to the network is advantageous, as djpipe will then not need to copy the data as it would if it were reading from *stdin*.

Security

There are several levels of security in the DeeJay system. We are concerned with security for the hosts being backed up, as well as for the data on the tapes.

Coback runs as the user *backup*, as does djback on each host. The backup user is placed in the *operator* group on each backee. In the usual fashion, the operator group is given read access to the raw and block devices for the filesystems which need backing up. Thus, the backees need not give out root access to their systems; they only allow the backup user from the coback host.[22]

When dj gets a connection, it looks up the address connecting to it, and compares it to the list of allowed hosts (from the *hosts.dj* file). Hostnames are matched using wild cards, so that access can be given to an entire group of machines easily. Access can be either for **read**, **write**, or **none**.

Interestingly, read access is stronger than write access. Write access simply means that the host can append a new saveset at the end of a tape. While this could use up all existing tapes (causing a denial of service), it does not compromise the security of the data which is already on the tapes. Since these are backup tapes, letting someone read a tape gives them access to all files on that tape — thereby circumventing the filesystems' own access systems. Therefore, read access is more crucial.

We give read access to those hosts which will need to do restorals. However, they also need the ability to mount and position a tape, and to read a table of contents. So we include these abilities in read access. The backup coordinator also needs this access, as well as being able to outdate a series, so we include that in read access too.[23] All backees are given write access.

Outstanding Problems

Our checkpointing scheme is too simplistic, and needs to be improved. We often find ourselves rewriting a checkpoint file to be what we really want, or running a makeup backup by hand.

For example, if a Monday night backup fails in the middle of running incrementals, we won't notice until we come in Tuesday morning. At this point, the machines are too busy to be running backups, so

[22]This might be one reason to run coback on a different host from dj — if another department wanted us to back up their hosts, they could run coback on one of their hosts and not have to trust our hosts (though this would require us to trust their host).

[23]Perhaps this should be refined for greater security.

we plan the makeup for that evening. Since Tuesday's backups will be ready to run then, we would end up running Monday's incrementals, Monday's fulls, Tuesday's incrementals, and then Tuesday's fulls. This would be a lot of extra work for little extra data, so we generally choose to skip the missed incrementals, simply running Monday's fulls and then the regular Tuesday night backups.

Another thing the checkpoint scheme doesn't cover is host-type errors. If one host is down when coback tries to run its backups, that host will be skipped while coback continues on with the rest of the hosts. This is certainly better than skipping everyone's backups due to one troublesome host. However, there is no checkpoint information indicating that that host was skipped. We currently depend on reading log files, noticing that the host was missed, and running a makeup by hand (or making up a checkpoint file, where possible).

A serious problem when the tape host crashes is that the reset of the SCSI bus at boot time usually causes the 8mm tape to rewind. This caused problems in early versions of DeeJay, since it would just start writing to the beginning of the tape, overwriting previous backups. We soon added some sanity checking, so every time DeeJay is about to start writing a new file to a tape (and some other times as well), it confirms that the position of the tape as reported using *ioctl*(2) and *mtio*(4) is what dj expects it to be. However, if the position is incorrect dj doesn't know what to do and leaves that tape unusable, requiring human intervention.

Given the way that coback directs one host after another to do its backups, each host depends on the host before finishing in a timely fashion. Due to the way SunOS implement sockets, if one host crashes in the middle of doing its backups, coback might wait forever for its connection to that host to close.[24] Sometimes, we would come back after a weekend and find coback still waiting for a host that had crashed days ago. To work around this, we made the server time out connections that were idle for too long. However, since coback uses rsh to run djback remotely, it still hangs when the backee crashes. Fixing this would require reimplementing *rcmd*(3N).[25]

Initially, we had a lot of problems with the hardware. Most of these were solved by cleaning the drives more often (we now run a cleaning tape every week) and installing a filter on the fan. We had a few problems with the original jukebox caused by the wear of yanking the drawer out to refill it — the rails eventually wore out and needed replacing. Also, removing the carousel was inconvenient. These problems should be solved with our newer IceBox — currently, the larger capacity means we rarely have to change tapes, but with its input/output arm and easy-load 10-tape magazines, we anticipate much easier tape swapping when our disk capacity eventually increases past the limits of the IceBox.

One rather infrequent problem we have is with writing EOF or HDR labels at the very end of the tape. When the SunOS *st*(4) driver sees that the end of the tape is being reached, the current *write*(2) system call returns an error. The process is then required to write a tapemark before any further data can be written. It can then write a trailer (EOV label). This works fine when the process is writing a data file to the tape (almost all of the time). However, when EOT is reached while a label is being written, writing a tapemark would end the label prematurely. A complete solution to this (given the current behavior of the st driver) requires backspacing over the previously written data block, writing an end of volume label, and then rewriting the overwritten data onto the next tape. This has not yet been implemented, but this bug has only occurred twice in the past year.

Another problem we expect to see is that of network contention. We have seen transfer rates of up to 4 megabits per second on some of our backups. Since our ethernet has a theoretical maximum of 10 megabits per second, two sets of backups running concurrently could swamp it (and we have four drives in our IceBox). Thus far, this has not been an issue, but we are looking out for it.

Future Extensions

There are several features which we would like to either add to the system or expand upon.

The way we handle the table of contents for a tape is not powerful enough. Ideally, the data about files on a backup should contain the modification time and owner of each file. Unfortunately, this data is not saved in the header of each dump, and so the restore program does not display it – scanning an entire dump as it is written would slow things down too much.

Since we do have some table of contents information saved, and a mechanism for accessing it, we would like to write a nice graphical interface for doing restorals. We were planning on using a NeXTStep browser which would represent the filesystem as a tree, with separate browsers for different savesets. Since we do very few restorals, we haven't allocated the time to this yet.

We would also like to improve the security of the system. While the host based scheme we are using is sufficient for now, we plan to use a Kerberos based authentication scheme in the future. If

[24] This is similar to the phenomenon when a host crashes and your xterm window doesn't disappear until you try to type in it.

[25] We would probably want the **SO_KEEPALIVE** option on rcmd's sockets.

we can verify who a requestor is, we can limit that person's access to only the tapes or savesets which he/she should have access to.

A secure authentication scheme will allow us to handle user tape mount requests. This will bring us back to where we were 10 years ago, when user tape mounts were supported on our DEC-20's.

We also need to expand our checkpointing scheme. Rather than just keep track of where things failed, we will write a list of what needs to be done. As backups are done, they will be removed from the list. This will make it very clear just what still needs to be done. During the night — some time after the original backup ran, an attempt could be made to recover from the checkpoint files. If some transient error occurred, then the backups could continue at this time.

Conclusion

One of the hard things about developing a system like this is testing it. The only way to see what breaks under a production load is by running in a production environment. However, as backups can be extremely important to the reliability of the systems, we did not want to compromise their integrity. As we slowly migrated different sets of backups to the system, the human workload progressively moved from that of our operators mounting tapes to having us babysit the backups. As things stabilized, we didn't need to watch the backups as closely, but we still needed to change tapes in the carousel, and the backups did require frequent tweaking. Currently, the backups are quite stable — we still get nightly mail about the backup runs, and things do still break occasionally, but for the most part things run completely unattended. At most, we sometimes need to restart a backup which has failed.

In addition, we have seen the amount of data being backed up grow from 4 central UNIX systems, with a total of 18 gigabytes of disk space between them, to 29 systems (including our first Novell system) holding about 40 gigabytes. Since there is much less wasted time (waiting for tapes to be mounted and unmounting tapes between every dump), and since we now stream the 8mm drives more efficiently than we did the 9-track drives, it takes us half the time to back up twice the data. So, our bottleneck has moved from the operators who used to mount the tapes (and keep the dump program waiting) to the network, which can't keep up with the amount of data we can pump across it.

Acknowledgments

Charlie C. Kim and Travis Lee Winfrey wrote our first UNIX backup scripts in 1986. Fuat C. Baran and Howie Kaye wrote the fancy version that scheduled backups according to the dumpdates file.

Thanks to Max Evarts, Ken Suh, and the rest of the Kermit Distribution crew for backing up our staff workstations for so long, until we got DeeJay really going.

Thanks to all the operators in the machine room, who mounted tapes for all those years, didn't get insulted when we bought an IceBox instead, and kept reminding us which way the bar codes go.

Thanks to the folks at BoxHill Systems Corporation, for selling us the Summus Jukebox, letting us trade it in for the IceBox, and letting us have both while we generalized our software some more. And especially to Vice President Chris Maio, who lugged parts up on the subway when our Jukebox needed fixing, and brought us a filter to keep the fan from sucking in dust.

Availability

The DeeJay package is available for anonymous FTP from **ftp.cc.columbia.edu**.

If you find DeeJay useful and would like to support the institution that has enabled us to make it freely available, we suggest that you send a voluntary contribution to Columbia University, as described in the documentation.

Author Information

Howie Kaye and Melissa Metz are Systems Programmers and Administrators in Columbia University's Academic Information Systems (AcIS). They each received an M.S. degree in Computer Science from Columbia — Howie in 1988 and Melissa in 1989. They are also known as two of the four authors of the **Columbia-MM** mail user agent. Their electronic mail addresses are howie@columbia.edu and melissa@columbia.edu. They can be reached via U.S. Mail at Watson Lab, 612 W. 115th St., New York, N.Y. 10025.

Dealing with Lame Delegations

Bryan Beecher – University of Michigan

ABSTRACT

The University of Michigan is a large, research university with over six thousand computers attached to six interconnected class B networks. The ownership and administration of these machines is widely distributed across the university, and consequently the U-M Domain Name System namespace is also quite distributed. While this keeps the workload low for any single campus hostmaster, it also introduces many possibilities for domain nameserver misconfiguration.

For some time we have run a version of the domain nameserver daemon, **named**, which includes Don Lewis' *lame delegation patch.* This patch uses syslog to generate an alert whenever *named* encounters a lame delegation, that is, an instance where a nameserver is listed as authoritative for a domain, but in fact is not performing service for that domain. We have taken this idea one step further by running a weekly job that collects these alerts, does its best to screen out any spurious ones, and then notifies the owner of the domain. This paper summarizes and discusses this work.

Background

The University of Michigan is a large and sprawling collection of TCP/IP networks, comprising half a dozen class B networks (141.211.0.0 through 141.216.0.0). The Domain Name System (DNS) entry for the University of Michigan is **UMICH.EDU**, and like the University's IP networks, it too is large and sprawling. Each college, institute, or campus-wide entity is entitled to a domain at the **UMICH.EDU** level. For example, the College of Literature, Science, and the Arts uses the **LSA.UMICH.EDU** domain. Departments within colleges then fall under that college's domain, and so the Math Department is known as **MATH.LSA.UMICH.EDU**. All in all, there are over six thousand hosts with entries in the DNS at U-M.

It is impractical to try to centralize the domain nameservice for this many machines. Consequently, many of the sub-domains of **UMICH.EDU** have been delegated to various system administrators across campus. This makes for a very distributed system where no single hostmaster has too much work. However, it also introduces great possibilities for errors when newly hired system administrators are forced to maintain their part of the DNS when they haven't had proper training. One of the most common errors introduced is that of a *lame delegation.*

The Problem

A *lame delegation* is an instance when a nameserver is listed as authoritative for a domain, but in fact, it is not performing service for that domain. A lame delegation is often caused at U-M when a hostmaster moves a domain nameserver from host A to host B without notifying the hostmaster of the parent domain. For example, say there are two sets of nameservers listed for **LSA.UMICH.EDU**. One set is listed in the **UMICH.EDU** domain zone file (where the domain is delegated) and the other set is listed in the **LSA.UMICH.EDU** domain zone file. Under normal operating conditions these two sets of nameservers should be identical, but if a hostmaster is careless and changes the local set without notifying his parent domain's hostmaster, they can get out of sync. In our example, if the listed nameservers for the **LSA.UMICH.EDU** domain are listed as **A.LSA.UMICH.EDU** and **B.LSA.-UMICH.EDU** in the **UMICH.EDU** zone file, but are listed as **C.LSA.UMICH.EDU** and **B.LSA.-UMICH.EDU** in the **LSA.UMICH.EDU** zone file, and the nameserver on **A.LSA.UMICH.EDU** stops serving the **LSA.UMICH.EDU** domain, then the **UMICH.EDU** zone file now contains a lame delegation.

Lame delegations are serious problems. At the very least, they cause the DNS to become much less efficient since query packets will be sent to hosts which are either not running a nameserver at all, or will be sent to nameservers which are not authoritative for a domain. In either case, the query will not be answered, and the sender will have to try another nameserver. In a more serious case, all of the nameservers listed for a domain will be lame delegations, and no queries can be answered! Interestingly, a site can be the victim of a serious lame delegation such as this without even being aware of it IF all of the resolvers on-site are configured to query a non-lame server, yet the servers listed for that site at its parent domain are all lame. Unfortunately, many excellent examples of this exist even today in the **IN-ADDR.ARPA** namespace. Here the scenario is that a site lists no nameservers for its **IN-ADDR.ARPA** domains(s) in the root servers due to ignorance or oversight, but their local nameservers do indeed serve their **IN-ADDR.ARPA** domain. And so, local queries all work since the packet is

coincidentally going to a nameserver which is authoritative for that **IN-ADDR.ARPA** domain, yet all non-local queries on that **IN-ADDR.ARPA** domain fail since there are no nameservers for it listed in the root servers.

The Solution

Don Lewis, one of the few people across the Internet contributing bug fixes and enhancements to BIND, released a patch which detects lame delegations. A lame delegation is detected when *named* forwards a query to a nameserver which was listed as an authoritative server, but the response is not marked as authoritative. A lame delegation causes a message like this to appear in the appropriate syslog log:

```
Jun 23 11:06:42 totalrecall
named[104]: Lame delegation to
'nadn.NAVY.MIL' received from
26.7.0.102 (purported server for
'NADN.NAVY.MIL') on query on name
[ward.nadn.navy.mil]
```

In this example, the namserver on totalrecall came across this lame delegation when it tried to resolve the name **WARD.NADN.NAVY.MIL**. At the time of this writing, there are three listed nameservers for the **NADN.NAVY.MIL** domains with the names: **NADN1.NADN.NAVY.MIL** (address 131.121.1.1), **USNA.USNA.NAVY.MIL** (26.7.0.102 and 128.56.1.1), and **NADN2.NADN.NAVY.MIL** (131.121.1.2). In this case, our nameserver forwarded the query on the name **WARD.NADN.NAVY.MIL** to a nameserver it was led to believe was authoritative for **NADN.NAVY.MIL**, yet in this case, that nameserver replied with non-authoritative data.

Since we started running a version of *named* with this patch, we found that we discovered hundreds of lame delegations each month. Furthermore, although some lame delegations were indeed problems with a domain nameserver at the University of Michigan, most were problems with domain nameservers located elsewhere in the Internet.

At first we ignored the non-local lame delegation messages and simply used a tool like *grep* to extract the lame delegations that were local. Yet, even this proved to be less effective than we hoped since many of the lame delegations proved to be transient problems, and there was little sense in alerting a hostmaster to a problem that no longer existed. We decided to build a tool which would screen out as many transient or spurious errors as possible,

and would then automatically alert the appropriate hostmaster via e-mail.

The Tool

We wrote a small shell script which we run out of *cron* once per week. The script is relatively short, and we thought the best way to present it would be to simple include an annotated version of it here. It is also available via anonymous ftp from terminator.cc.umich.edu. It can be found in /dns/lamers.sh.

The Script

The first part of the script sets up our path, identifies the location of the log file, identifies the file containing the lame delegation message we send to people, and identifies some temp files. Notice that some temp files are located on /usr/tmp rather than /tmp. Some of these files can get big as the script runs, and we found that we would fill up /tmp (which is on the / partition on our machines).

```
#!/bin/sh
PATH=:/bin:/usr/bin:/usr/ucb:/usr/local/bin
LOGFILE=/usr/spool/log/named
MAILMSG=/usr/tmp/mailmsg$$
LAMERS=/usr/tmp/lamers$$
MSGFILE=/usr/local/bin/lamer-message
LAMEREPORT=/tmp/.lamereport$$
WEEKFILE=/usr/tmp/week$$
```

The next part contains some standard information we put in anything we make available to the Internet community: A standard copyright notice, the author's name, the last change date, and some notes about the tool.

The note below also lies a little bit. You can make use of this script even if you do not have either *query* or *host*. *query* is a simple program which queries nameservers using the resolver routines available in the C library. Unlike some other packages, it does not include external resolver routines to link in; it uses the same code that the other software on the machine uses. *host* is a short shell script around *query* which, when given an IP address as an argument, returns the associated host name. Tools such as *dig* and *nslookup* would also do just fine, although the shell script would have to be modified appropriately to handle their output instead.

```
# ------------------------------------------------------------
```

```
#  Copyright (c) 1991 Regents of the University of Michigan.
#  All rights reserved.
#
#  Redistribution and use is permitted provided that this notice
#  is preserved and that due credit is given to the University of
#  Michigan. The name of the University may not be used to endorse
#  or promote products derived from this software without specific
#  prior written permission. This software is provided "as is"
#  without express or implied warranty.
#
#  Lame delegation notifier
#  Author:  Bryan Beecher
#  Last Modified:   6/25/92
#
#  To make use of this software, you need to be running the
#  University of Michigan release of BIND 4.8.3, or any version
#  of named that supports the LAME_DELEGATION patches posted to
#  USENET.  The U-M release is available via anonymous ftp from
#  terminator.cc.umich.edu:/unix/dns/bind4.8.3.tar.Z.
#
#  You must also have a copy of query(1) and host(1).  These
#  are also available via anonymous ftp in the aforementioned
#  place.
# ------------------------------------------------------------
#  handle arguments
# ------------------------------------------------------------
#       -d <day>
#       This flag is used to append a dot-day suffix to the LOGFILE.
#       Handy where log files are kept around for the last week
#       and contain a day suffix.
#       -f <logfile>
#       Change the LOGFILE value altogether.
#       -w
#       Count up all of the DNS statistics for the whole week.
#       -v
#       Be verbose.
#       -t
#       Test mode.  Do not send mail to the lame delegation
#       hostmasters.
# ------------------------------------------------------------
```

For a given service that we provide on one of our machines, we maintain one week's worth of log files, broken into eight log files: one for each previous day of the week plus one for the current day. For example, the current log file for *named* would be located in /var/log/named. And if it happens to be Tuesday, then yesterday's log file would be located in /var/log/named.Mon, the previous day's would be located in /var/log/named.Sun, and last week's log file would be in /var/log/named.Tue .

The -f and -t flags exist mainly for debugging. Before we unleashed this upon the Internet community, we wanted to make sure that we wouldn't be generating tons of unwanted mail. The -f flag is handy when you'd like to hand-generate your own lame delegation data and then use it with this script. The -t flag performs all the usual work, except it does NOT notify the hostmaster via e-mail. It does, however, still build a list of who it would have mailed had the -t flag not been specified.

```
VERBOSE=0
TESTMODE=0
while [ $# != 0 ] ; do
  case "$1" in
    -d)
    LOGFILE=$LOGFILE"."$2
```

```
    shift
    ;;
    -f)
    LOGFILE=$2
    shift
    ;;
    -w)
    cat $LOGFILE* > $WEEKFILE
    LOGFILE=$WEEKFILE
    ;;
    -v)
    VERBOSE=1
    ;;
    -t)
    TESTMODE=1
    ;;
  esac
  shift
done
```

We added the following line so that the script would clean up after itself if it was killed during a run.

```
#-------------------------------------------------------------------------
#  Clean up and exit on a HUP, INT or QUIT
#-------------------------------------------------------------------------
trap "rm -f $LAMERS $MAILMSG $LAMEREPORT $WEEKFILE ; exit" 1 2 3
```

The first thing we do is search the log to see if any lame delegations were detected. We toss out lines with an asterisk on them since those tended to be lame delegations of the form "server xxx.xxx.xxx.xxx is a lame delegation for domain *". We really aren't able to do anything with a message like that, and it isn't clear to us exactly how those are getting generated either. We also down-case everything at this point so its easier to parse and handle later.

After the initial pruning we strip off the domain name and nameserver's IP address from the line in the log file. We sort those, toss out duplicates, and write the results to a temp file. If the temp file is non-empty, we know that we found some lame delegations to handle.

```
#-------------------------------------------------------------------------
#  See if there are any lamers
#-------------------------------------------------------------------------
grep "Lame" $LOGFILE | tr A-Z a-z | grep -v "*" | awk '{
        print substr($16, 2, length($16) - 3), $12 }' |
        sort | uniq | awk '{
                printf("%s %s\n", $1, $2)
}' > $LAMERS

if [ ! -s $LAMERS ] ; then
        exit 0
fi

if [ $VERBOSE -eq 1 ] ; then
        echo "Found" `awk 'END { print NR }' $LAMERS` "lame delegations"
fi
```

The following message mentions *potential* lame delegation because we've often found that there are cases when the lame delegation patch flags a server as lame even though it does not appear to be lame by the time this script runs. It isn't clear to us if the culprit here is the lame delegation patches (which are flagging innocent nameservers as lame), *named* itself (which is giving bogus information to the lame delegation code), or if there are simply many transient errors in the domain name system.

At this point each lame delegation is identified by a unique (nameserver, domain) pair.

```
#  There were lamers; send them mail
touch $LAMEREPORT
NAME=""
while read DOMAIN IPADDR ; do
        #-------------------------------------------------------------
        # Echo args if verbose
        #-------------------------------------------------------------
        if [ $VERBOSE -eq 1 ] ; then
                echo $IPADDR "is a potential lame delegation for" $DOMAIN
        fi
```

The next thing we do is lookup the SOA record for the domain. We do this so that we can fetch an ostensibly official e-mail address to which to send the mail. In our experience the e-mail address listed in an SOA record often is syntactically incorrect, or contains some unusable address. The script isn't too careful about this, and so we end up seeing a fair number of bounces which we then handle ourselves.

```
#-------------------------------------------------------------
# Lookup the SOA record form $DOMAIN.  A really broken name
# server many have more than one SOA for a domain, so exit
# after finding the first one.  Send it to the local hostmaster
# if we cannot find the proper one.
#-------------------------------------------------------------
if [ $VERBOSE -eq 1 ] ; then
        echo "Looking up the hostmaster for $DOMAIN"
fi
HOSTMASTER='query -h $DOMAIN -t SOA 2> /dev/null | \
                    awk '/mail addr/ { print $4 ; exit }' | sed -e 's/./@/''
NAME='host $IPADDR 2> /dev/null'
if [ -z ""$HOSTMASTER ] ; then
        if [ -z ""$NAME ] ; then
                HOSTMASTER="hostmaster"
        else
                HOSTMASTER="postmaster@"$NAME
        fi
fi
```

This is one of the tests we make to weed out the spurious lame delegations. There have been cases where a parent server has listed another nameserver as authoritative for a domain, yet that actual server does not report itself as authoritative for a domain. This is still a problem, and in the future we should do something better here than just continue (e.g., send mail to the parent domain telling them about the problem).

```
#-------------------------------------------------------------
# Find the name associated with IP address $IPADDR.  Query
# the nameserver at that address:  If it responds listing
# itself as a domain namserver, then it is lame; if it isn't
# in the list, then perhaps the lame delegation alert was
# spurious.
#-------------------------------------------------------------
if [ $VERBOSE -eq 1 ] ; then
        echo "Making sure that $IPADDR is listed as a NS for $DOMAIN"
fi
if [ -n ""$NAME ] ; then
        query -n $IPADDR -h $DOMAIN 2>&1 | grep "domain name" | \
                                                grep -i $NAME > /dev/null
        if [ $? -eq 1 -a $VERBOSE -eq 1 ] ; then
                echo $NAME does not seem to be a nameserver for $DOMAIN
                continue
        fi
fi
```

We query the listed nameserver twice. Even in the case of a lame delegation, it may return with authoritative data on the first query since it may have just made the query to an authoritative nameserver. If it is a lame delegation, then the second query will be from the nameserver's cache rather than from its authoritative data, and so the **aa** (authoritative answer) header flag will be missing. Some really malformed answers set all of the flags, and so if an unusual one is set, like **tc** we consider the answer to be invalid.

```
#-------------------------------------------------------
# If the delegation is no longer lame, don't send mail.
# We do the query twice; the first answer could be authori-
# tative even if the nameserver is not performing service
# for the domain.  If this is the case, then the second
# query will come from cached data, and will be exposed
# on the second query.  If the resolver returns trash, the
# entire set of flags will be set.  In this case, don't
# count the answer as authoritative.
#-------------------------------------------------------
if [ $VERBOSE -eq 1 ] ; then
        echo "Making sure that $IPADDR is not providing authoritative data now"
fi
query -n $IPADDR -h $DOMAIN > /dev/null 2>&1
query -n $IPADDR -h $DOMAIN 2>&1 | grep header | grep aa | \
                                                  grep -v tc > /dev/null
if [ $? -eq 0 ] ; then
  if [ $VERBOSE -eq 1 ] ; then
    if [ -n ""$NAME ] ; then
            echo $NAME seems to be serving $DOMAIN OK now
    else
            echo $I seems to be serving $DOMAIN OK now
    fi
  fi
  continue
fi
```

If we've reached this point, then we probably have a genuine lame delegation. We then use *sed* to do a quick substitution on a copy of the message we send out to mark in the proper domain name and namserver IP address. We then mail off the message using the name "dns-maintenance@umich.edu" in the "From:" line. That way if it bounces (or gets a reply), the message will not go to root's mailbox.

```
#-------------------------------------------------------
# Notify the owner of the lame delegation, and also notify
# the local hostmaster.
#-------------------------------------------------------
if [ $TESTMODE -eq 0 ] ; then
  if [ $VERBOSE -eq 1 ] ; then
    echo "Sending to $HOSTMASTER about lame server $IPADDR for domain $DOMAIN"
  fi
  echo "To: " $HOSTMASTER > $MAILMSG
  echo "Subject: $IPADDR seems to be a lame delegation for $DOMAIN" >> $MAILMSG
  cat $MSGFILE >> $MAILMSG
  sed -e "s|%DOMAIN%|$DOMAIN|" -e "s|%SERVER%|$IPADDR|" $MSGFILE |
              /usr/lib/sendmail -t -fdns-maintenance
fi
echo $HOSTMASTER $DOMAIN $IPADDR >> $LAMEREPORT
done < $LAMERS
```

Now that we have processed all of the lame delegation messages, we're just about done. The last action we take is to send a single message to the local hostmaster at U-M listing all of the lame delegations. The message - a short lame delegation report - lists one line per lame delegation found. On that line are the domain, the nameserver, and the e-mail address that we used when sending the message.

```
#-------------------------------------------------------
# No news is good news
```

```
#-------------------------------------------------------------
if [ -s $LAMEREPORT ] ; then
        Mail -s "Lame report" hostmaster@umich.edu < $LAMEREPORT
fi
rm -f $LAMERS $MAILMSG $LAMEREPORT $WEEKFILE
```

Conclusions

We've been using this tool to notify hostmasters of lame delegations for about one year now. So far the results have been very encouraging. Except for some really hard-core lamers, most sites who are flagged one week as having a lame delegation do not get flagged the following week. Unfortunately, the number of lame delegations discovered each week doesn't seem to be going down either. We've been told by the folks at NSFNET that domain packets account for about 10% of the backbone T3 traffic; it would be interesting to conduct an experiment to try to discover exactly how much of that is due to misconfigured nameservers.

Response from hostmasters receiving mail from this script has generally been very favorable. Comments run the gamut from "Thanks for the heads up. We don't mind at all getting the mail." to "Quit sending us this message...". Most of the unfavorable replies come from people who don't understand why they're getting the message, and after a friendly phone call or follow-up message, they are usually eager to cooperate.

All of the software mentioned above and the e-mail template that we use are available via anonymous ftp from terminator.cc.umich.edu:/dns.

Author Information

After receiving his MS in Computer Engineering from the University of Michigan in 1988, Bryan Beecher could not bring himself to leave academic life for the hardships of a real job, and so has spent the past four years handling domain nameservice and teaching UNIX system administration courses for U-M. The U-M distribution of BIND 4.8.3 contains many features he has added, such as Shuffle Address (SA) records, query logging and report generation, and, of course, the lame delegation tools. Most recently, he has begun running U-M's USENET news server *(destroyer),* and is part of the team creating better tools for X.500-based directory services. He can be reached via e-mail at bryan@umich.edu.

Majordomo: How I Manage 17 Mailing Lists Without Answering "-request" Mail

D. Brent Chapman – Great Circle Associates

ABSTRACT

Majordomo is a perl program written to handle routine administration of Internet mailing lists with as little human intervention as possible. Modeled after the Listserv implementations common on BITNET (but unfortunately rare on the Internet), it automates the administration of mailing lists by allowing users to perform the most frequent operations ("subscribe" and "unsubscribe") themselves, while allowing the list owners to either "approve" each of these operations (or initiate them on behalf of a user), or merely monitor them as they are automatically approved. It also automates response to certain other common queries from users, such as "what lists are served by this Majordomo server?", "what is the topic of list 'foobar'?", "who is already on list 'foobar'?", and "which lists managed by this Majordomo server am I already on?".

Majordomo allows individual list owners to manage their own lists (subscribe and unsubscribe users, and change the general information message for their list) without any action by the overall Majordomo owner. It serves both "open" lists (where users can add themselves to the list, and the list owner is merely informed of this action) and "closed" lists (where a subscription request from a user generates an approval request from the Majordomo server to the list owner, who can then either approve or ignore the request).

Finally, all interactions with Majordomo by both users and list owners take place totally by electronic mail, so users and list owners do not require login access (nor even direct TCP/IP connectivity) to the machine Majordomo is running on, and no special client software is required.

Introduction

Anyone who has ever managed a significant electronic mailing list by hand (which is, on the Internet at least, the usual method) knows how much time it takes to process the endless requests from users of the form "please subscribe me to your list", "please unsubscribe me from your list", "please tell me about your list", "please tell me if I'm already on your list", and so forth. It's a time-consuming, boring, repetitive task; just the sort of thing that's a perfect candidate to be automated.

When SAGE (the System Administrators Guild, a USENIX Special Technical Group) was formed, the founding members decided to establish over a dozen mailing lists for various purposes (one for the board of directors, one for each of the 16 initial working groups, one the chairs of all the working groups, and so forth). The USENIX Association volunteered the USENIX.ORG machine as a home for these mailing lists, but didn't have the staff resources to set up and operate the mailing lists. I volunteered to act as Postmaster for SAGE, and handle all the mailing lists. As an independent consultant, my schedule is rather erratic, and I don't have a company paying my salary while I pursue volunteer work like this; thus, I wished to automate the job as much as possible, so that I could provide a high level of service to the users (including fast turnaround on their requests) while spending as little time as possible in the long run on administrivia. A BITNET-style Listserv seemed to be an appropriate solution, so I started investigating alternatives.

Defining the Problem

The first step was to identify just what functionality I desired. First and foremost, I wanted something that would handle routine "subscribe" and "unsubscribe" requests automatically, with no human intervention required for routine requests (though I wanted to give the owner of a given list the option of passing judgement on all subscription requests, if they so desired). Second, I wanted something that could easily handle many mailing lists simultaneously; I had 17 to begin with, and I was sure that more would be added as time passed. Third, I wanted something that could automatically handle other user requests (such as "what lists are available?", "please tell me about list 'foobar'", and "which of your lists am I on?") that, while less common than "subscribe" and "unsubscribe", still occur relatively frequently.

The first thing I did was look around for suitable publicly available software that might already exist, or that might be easily adapted to my needs. Searches of the common Internet software archives, queries to the "Archie" anonymous FTP indexing

service, and email to certain acquaintances who I thought might know of such software produced two results: an implementation of the BITNET Listserv written in C for UNIX (from the comp.sources.unix archives), and several different programs named "listserv" written in perl.

I first examined the BITNET Listserv C package from the comp.sources.unix newsgroup archives. It looked like it would do most of what I wanted, but it also looked like it did a lot of things I didn't really care about (there appeared to be features for coordinating activities between multiple Listserv servers on different machines, for instance). It appeared to be rather short on documentation, and what documentation there was seemed to assume that the reader was already familiar with BITNET Listserv implementation and operation. All in all, it looked like it would be a real headache for me to install, configure, and maintain, since I'm *not* familiar with BITNET Listserv implementation and operation.

The next things I looked at were several perl scripts from a variety of sources that were supposedly Listserv-like servers. Some of these scripts were pointed out to me by folks on the net who knew I was looking for such a thing, and I found others by searching through Archie for "listserv". Unfortunately, these various scripts all turned out to be more what I'd call "archive servers" than "listserv" implementations; they were written to automate retrieval of files from archives via email, for folks who don't have access to anonymous FTP. When I examined one of these scripts that claimed to support "subscribe" and "unsubscribe" requests, I found that what it did with such requests was forward them by email to the mailing list owner for manual processing; this was exactly what I was trying to avoid!

In the end, I decided to implement my own version of Listserv, so that I could get exactly what I wanted. The name for my software was provided by Eliot Lear of Silicon Graphics, Inc.; he suggested "majordomo", which the dictionary defines as "a person who speaks, makes arrangements, or takes charge for another", and which seems perfectly appropriate given the nature of the software.

Designing a Solution

My first step in designing a solution was to decide on the general approach I was going to take. First, I decided that all routine interactions with Majordomo would take place asynchronously via email. Second, since the software was going to spend most of its time parsing emailed instructions, processing text files (the actual mailing lists) according to those instructions, and generating emailed responses to users, I wanted to write it in a language well-suited for that task; perl seemed the natural choice.

In the Majordomo world model, there are three types of people: users (without any special privileges), mailing list owners, and the owner of the Majordomo server itself. Interactions with users take place strictly by email; the user mails a set of requests to Majordomo, and Majordomo processes those requests and sends back appropriate replies. Interactions with list owners also take place strictly by email, but a list owner can do a few things that a normal user can't; the commands that are restricted to list owners are protected with a per-list password (though it's very weak password protection, since the password is passed in the clear through the email; the goal is not absolute security, but to avoid people making a nuisance of themselves by abusing the Majordomo server). The Majordomo owner is the person responsible for maintaining the Majordomo server itself, and for performing tasks such as creating new mailing lists to be served by Majordomo.

The software needs to support multiple mailing lists, each owned by different individuals. Some owners wish to approve all "subscribe" requests for their list (a "closed" list), while other owners wish routine "subscribe" requests to be approved automatically (an "open" list), with notification to the owner.

Command	Description
subscribe *list* [*address*]	Subscribe yourself (or *address*, if specified) to *list*
unsubscribe *list* [*address*]	Unsubscribe yourself (or *address*, if specified) from *list*
which [*address*]	Find out which lists you (or *address*, if specified) are on
who *list*	Show the members of *list*
info *list*	Show the general introductory information for *list*
lists	Show the lists handled by this Majordomo server
help	Retrieve a help message, explaining these commands
end	Stop processing commands (useful if your mailer automatically adds a signature to your messages)

Figure 1: Majordomo user commands

Routine "unsubscribe" requests are approved automatically, with notification to the list owner, for both open and closed lists. Owners have a way (the "approve" command) to approve all "subscribe" requests on closed lists, as well as non-routine "subscribe" and "unsubscribe" requests on open lists. A "non-routine request" is one that affects a different address than the request appears to originate from; for instance, a request from "joe@foobar.com" to subscribe or unsubscribe "alice@foobar.com" is a non-routine request. All non-routine requests (on both open and closed lists) are forwarded to the list owner for approval.

Majordomo accepts the commands shown in Figure 1 from any user. In addition, Majordomo accepts the password-protected commands shown in Figure 2, which are for use by list owners to manage their list. Authentication is based solely on knowledge of the password for the list in question; no attempt is made to check that the address of the person issuing the command is the same as the address of the list owner. As mentioned earlier, the goal of the minimal security in Majordomo is to prevent anti-social people from making a nuisance of themselves; I don't make any claims that the security is particularly strong.

A side benefit of authentication by password is that the owner can manage their list from any of their accounts; they don't have to always use the same account on a certain machine, for instance. The "owner" of a given list could in fact be an alias for multiple people, any of whom could approve requests for the list. Because the owner of a list is always notified of successful "subscribe" and "unsubscribe" requests concerning their list, even if the owner initiated those requests on behalf of a user, multiple owners would automatically be kept up to date on each other's actions concerning the list.

Note that the "approve" command is simply "approve *password*" prepended to a "subscribe" or "unsubscribe" request. This simplifies command processing; in handling an "approve" message, the command processor checks that the password is correct for the list being acted on, then recursively processes the "subscribe" or "unsubscribe" command with a flag set that tells the processor that the operation is pre-approved and should simply be carried out, even if it is a non-routine request. The right way to think about "approve", by the way, is that the list owner is telling Majordomo "I approve this command; just do it!", not "I approve this request you sent me earlier". Majordomo doesn't keep track of outstanding requests; when an "approve" command comes in from a list owner, Majordomo doesn't check to see that the owner is approving something Majordomo had previously requested, or anything like that. A list owner can thus issue "approve" commands on behalf of a user (to drop a dead account from the list, for instance) without any prior action by the user.

An important distinction that many people misunderstand is the difference between managing a mailing list, and managing the traffic on a mailing list. Managing a mailing list (which is what Majordomo does) means exactly that: managing a list of names. Managing the traffic on a mailing list (which is commonly called "moderating" the mailing list) means either automatically or manually reviewing each message that is submitted for the list, then either forwarding it to the list (perhaps after header or content editing, depending on the nature of the mailing list) or discarding it. The changes made to messages before forwarding them to such a moderated mailing list can be as simple as rewriting the headers of the message to arrange for errors to come back to the list owner, or as complex as completely rewriting the body of the message to preserve the anonymity of the originator. Editorial policies (such as only forwarding messages to the list that were sent by a member of the list, and refusing messages from "outsiders") might also be enforced automatically or manually. All of this is outside the scope of Majordomo; all Majordomo does is maintain the file containing the list of email addresses. How that list is used (whether it is simply included as an alias in the `/etc/aliases` file, or used by a forwarding that enforces a "no messages from non-members policy" as described above, or whatever) is not something for Majordomo to determine.

Implementing the Proposal

Once I had more or less decided what I wanted to implement and how, I sat down to the nitty-gritty details of getting it done. It took about 2 days of concentrated work to write the core of the program, followed by a test installation and another couple of

Command	Description
approve *password* {subscribe | unsubscribe} *list address*	Approve a non-routine subscribe or unsubscribe request concerning *list*
newinfo *list password*	Provide a new "info" message for *list*, to be sent in response to "info" and "subscribe" requests
passwd *list old-password new-password*	Change the password for *list*

Figure 2: Majordomo list owner commands

days of on-again, off-again testing and enhancement. All told, I spent about 20 hours on the project, and ended up with about than 600 lines of perl code that implemented almost all the features listed above (I didn't implement "which" and "unsubscribe" until a couple of weeks later). This was the version that was initially installed on USENIX.ORG to run the SAGE mailing lists in late June, 1992. Over the next couple of weeks, I spent another 20 or so hours implementing the remaining commands, fixing minor bugs, and generally cleaning up the program. I've continued to make minor enhancements since then. Today, the program stands at 815 lines of perl code, not including libraries.

While writing Majordomo, I made extensive use of other people's work that had been previously released on the net, including software to process mail headers and perform file locking. From one of the perl archives on the Internet, I obtained a perl package called "mailstuff.pl" (written by Gene Spafford) which parses RFC822 mail headers into perl associative arrays for easy processing; with a few minor modifications, it was just what I needed to handle all the mail header processing for Majordomo.

I needed a safe way for Majordomo to lock files while editing them (adding or deleting users on a mailing list, or changing the "info" file for a list, for instance), to prevent multiple Majordomo processes from tripping each other up. I was familiar with Erik Fair's "shlock" program, which is provided in the NNTP distribution as a file locking mechanism for use in shell scripts, and knew it would provide the kind of locking I wanted; porting the code from a stand-alone C program to a 150-line perl package was a relatively simple matter. The biggest problem I encountered was that the C code used "goto" to break out of nested command logic when exceptions occurred; unlike some, I don't dogmaticly object to "goto" on general principles, but this particular usage of "goto" simply isn't supported in perl.

Other complications included addressing and appropriate case sensitivity. It was slightly tricky to get all the "To:" and "From:" addresses correct on mail generated by Majordomo, so that replies to commands and requests for approval from Majordomo went to the right place, and could themselves be replied to with appropriate results. It was also tricky to get certain things to be case sensitive (passwords, for example), and other things to be case insensitive (email addresses, mailing list names, and commands, for instance); further, some case insensitive items (such as mailing list names) need to be smashed to lower case before use, while others (such as email addresses) need to be preserved in mixed case and merely compared in a case insensitive manner.

Because it needs to edit files (the mailing lists, the "info" files for each list, and so forth), I decided that Majordomo needed to run setgid to a specially-created group which would have appropriate permissions on those files. Perl includes a nifty dataflow-tracing feature (commonly known as "taintperl") that is automatically activated when a perl script is run setuid or setgid; this feature attempts to ensure that the script doesn't do anything "dangerous". The perl on-line manual page describes this feature:

> When perl is executing a setuid script, it takes special precautions to prevent you from falling into any obvious traps. (In some ways, a perl script is more secure than the corresponding C program.) Any command line argument, environment variable, or input is marked as "tainted", and may not be used, directly or indirectly, in any command that invokes a subshell, or in any command that modifies files, directories or processes. Any variable that is set within an expression that has previously referenced a tainted value also becomes tainted (even if it is logically impossible for the tainted value to influence the variable).

While this is certainly a valuable feature of perl, I wasn't able to get Majordomo to function because of it. I spent many hours trying to make "taintperl" happy before I gave up and wrote a simple C "wrapper" program that sets the real UID and GID to the effective UID and GID before executing the Majordomo perl script, thus not activating the "taintperl" feature. This is almost certainly *not* the right thing to do; at some point, I need to go back and figure out how to make Majordomo work under "taintperl". Particularly since I'm bypassing the "taintperl" security features, Majordomo makes a special effort to validate user input (email addresses and mailing list names, for instance) and ensure that it doesn't contain anything dangerous (a command

```
$whereami = "GreatCircle.COM";
$whoami = "Majordomo@$whereami";
$whoami_owner = "Majordomo-Owner@$whereami";
$homedir = "/usr/local/majordomo";
$listdir = "$homedir/Lists";
$log = "$homedir/Log";
```

Figure 3: Sample `/etc/majordomo.cf` file

like "|uudecode" in an email address or an absolute path name like "/etc/passwd" as a mailing list name) before using that input to interact with the operating system (by opening files by that name, and so forth).

The title of this paper states that I don't answer "-request" mail (that is, mail people send to "*list*-request" with requests concerning *list*). While that's true, *something* has to answer "-request" mail. Mail sent to "*list*-request" can't simply be forwarded to Majordomo for processing, since it almost certainly doesn't contain commands that Majordomo would understand. A simple little perl script called "request-recording" (abbreviated as "request-rec" in Figure 4) answers the "-request" mail for each mailing list, and sends back a message (customized to the list in question) telling the user how to use Majordomo to subscribe to the list, get information about the list, or get a copy of Majordomo's help file; in addition, instructions are provided on how to reach a human being, just in case.

Configuring Majordomo

At startup, Majordomo reads a configuration file (as specified by the "MAJORDOMO_CF" environment variable or on the command line, or "/etc/majordomo.cf" by default) that provides site-specific information, including the name of the site, who mail from Majordomo should appear to be from, where Majordomo's supporting programs are located, where the lists Majordomo manages are located, and where Majordomo's log is located. Figure 3 shows a sample Majordomo configuration file. All Majordomo-managed files (the lists themselves, and the "info" and "password" information for those lists) are kept in a directory specified by the "$listdir" variable in the configuration file. Each mailing list is kept in a file in the $listdir directory that is exactly the name of the mailing list. Mailing list names may contain only lower case letters, numbers, "-", and "_". The lists Majordomo thinks it manages are the files in $listdir whose names meet these criteria for mailing list names. There is no specific "list of lists" in a file anywhere; thus, creating a new list for Majordomo to manage merely involves creating a new file with appropriate permissions in $listdir and creating appropriate entries in either /etc/aliases or /usr/lib/aliases to use that file.

Several auxiliary files may be associated with each list in $listdir. The password for *list* is contained in the file "*list*.passwd". The descriptive info for *list* (which will be returned in response to a "info *list*" or "subscribe *list*" command) is in "*list*.info". The existence of a file called "*list*.closed" indicates that *list* is a "closed" list, and that all "subscribe *list*" requests must be approved by the list owner. Note that the names of these auxiliary files are invalid mailing list names, because they contain a "."; that's how Majordomo differentiates the mailing list files from the auxiliary files.

Majordomo is closely tied to the /etc/aliases or /usr/lib/aliases file. A number of aliases are required for the Majordomo server itself, as well as for each of the lists managed by Majordomo. Figure 4 shows sample entries for the /etc/aliases file on a machine using Majordomo to run two lists ("open-list" and "closed-list"). The "-approval" alias is where Majordomo will send requests for approval for actions concerning a list. The "owner-" alias is not used by Majordomo, but is used by Sendmail to notify the owner of a mailing list of problems with that mailing list (bounced messages, and so forth; see the Sendmail documentation for more information). The "owner-" and "-approval" aliases could point to different people; each could also expand to multiple people.

Using Majordomo

To use Majordomo, a user sends commands as an email message to the address the Majordomo server is configured to recognize (for the sample configuration in Figure 3, the address is "Majordomo@GreatCircle.COM"). For instance, to find out what lists are served by Majordomo@GreatCircle.COM, a user named "Jane@Somewhere.ORG" might send the following

```
majordomo: "|/usr/local/majordomo/wrapper /usr/local/majordomo/majordomo"
owner-majordomo: brent

open-list: :include:/usr/local/majordomo/Lists/open-list
open-list-request: "|/usr/local/majordomo/wrapper /usr/local/majordomo/request-rec open-list"
open-list-approval: joe@foobar.com
owner-open-list: joe@foobar.com

closed-list: :include:/usr/local/majordomo/Lists/closed-list
closed-list-request: "|/usr/local/majordomo/wrapper /usr/local/majordomo/request-rec closed-list"
closed-list-approval: bob@elsewhere.edu
owner-closed-list: bob@elsewhere.edu
```

Figure 4: Sample /etc/aliases entries

message:

```
From: Jane@Somewhere.ORG
To: Majordomo@GreatCircle.COM

lists
```

The "Subject:" line of a message, if any, is ignored by Majordomo, so there's no harm in leaving it out. Jane would receive a message like this in response to her query:

```
From: Majordomo@GreatCircle.COM
To: Jane@Somewhere.ORG
Subject: Majordomo results

>>>> lists
Majordomo@GreatCircle.COM serves the
following lists:

    majordomo-announce
    majordomo-users

Use the 'info <list>' command to get
more information about a specific list.
```

Upon receiving this, Jane might wish to find out more about each of these lists. She could send the following request:

```
From: Jane@Somewhere.ORG
To: Majordomo@GreatCircle.COM

info majordomo-announce
info majordomo-users
```

In return, Majordomo would respond with:

```
From: Majordomo@GreatCircle.COM
To: Jane@Somewhere.ORG
Subject: Majordomo results

>>>> info majordomo-users
This list is for discussions (including
bug reports, enhancement reports,
and general usage tips) concerning
the Majordomo mailing list manager.
...

>>>> info majordomo-announce
This list is for announcements of new
releases of the Majordomo mailing
list manager.
...
```

If Jane wishes to subscribe to one of the lists (say, the majordomo-users list), she would send the following request:

```
From: Jane@Somewhere.ORG
To: Majordomo@GreatCircle.COM

subscribe majordomo-users
```

In return, she would receive two messages. The first is a standard Majordomo response:

```
From: Majordomo@GreatCircle.COM
To: Jane@Somewhere.ORG
Subject: Majordomo results

>>>> subscribe majordomo-users
Succeeded.
```

The second is "welcome" message with specific information concerning the list (note that it also includes the same information that an "info" command on the list would return). This message goes to the subscribed address, not the address the request was made from (though in this case those are the same; since Jane didn't specify an address to subscribe, it defaulted to the address the request was made from):

```
From: Majordomo@GreatCircle.COM
To: Jane@Somewhere.ORG
Subject: Welcome to majordomo-users

Welcome to the majordomo-users mailing list!

If you ever want to remove yourself
from this mailing list, send the
following command in email to
"Majordomo@GreatCircle.COM":

    unsubscribe majordomo-users \
        Jane@Somewhere.ORG

Here's the general information for the
list you've subscribed to, in case you
don't already have it:

This list is for discussions (including
bug reports, enhancement reports,
and general usage tips) concerning
the Majordomo mailing list manager.
...
```

At the same time, the owner of the list (through the "majordomo-users-approval" alias in the `/etc/aliases` file on the Majordomo machine) would receive the following notification of a new user:

```
From: Majordomo@GreatCircle.COM
To: majordomo-users-approval@GreatCircle.COM
Subject: SUBSCRIBE majordomo-users

Jane@Somewhere.ORG has been
added to majordomo-users.
No action is required on your part.
```

If Jane wanted to subscribe some other address to majordomo-announce (the email address "SysStaff@Somewhere.ORG", for instance, so that all members of the system staff would receive announcements concerning Majordomo), she could submit the following request:

```
From: Jane@Somewhere.ORG
To: Majordomo@GreatCircle.COM

subscribe majordomo-announce \
        SysStaff@Somewhere.ORG
```

This would cause the following message to be returned to Jane:

```
From: Majordomo@GreatCircle.COM
To: Jane@Somewhere.ORG
Subject: Majordomo results

>>>> subscribe majordomo-announce \
        SysStaff@Somewhere.ORG
Your request to Majordomo@GreatCircle.COM:

    subscribe majordomo-announce \
        SysStaff@Somewhere.ORG

has been forwarded to the owner of the
"majordomo-announce" list for approval.
This could be for any of several reasons:

    You might have asked to subscribe to a
    "closed" list, where all new additions
```

```
    must be approved by the list owner.

    You might have asked to subscribe or
    unsubscribe an address other than
    the one that appears in the headers
    of your mail message.

When the list owner approves your request, you
will be notified.

If you have any questions about the
policy of the list owner, please contact
"majordomo-announce-approval@GreatCircle.COM".
```

At the same time, Majordomo sends the following message to the mailing list owner:

```
From: Majordomo@GreatCircle.COM
To: majordomo-announce-approval@GreatCircle.COM
Subject: APPROVE majordomo-announce

Jane@Somewhere.ORG requests that you
approve the following:

    subscribe majordomo-announce \
        SysStaff@Somewhere.ORG

If you approve, please send a message
such as the following back to
Majordomo@GreatCircle.COM (with the
appropriate PASSWORD filled in,
of course):

    approve PASSWORD subscribe \
        majordomo-announce SysStaff@Somewhere.ORG

If you disapprove, do nothing.
```

If the list owner sends such an "approve" command back to Majordomo, and the password is the correct password for the list in question, then the addition will take place. The address being subscribed (SysStaff@Somewhere.ORG, in this case) will receive a standard "Welcome to majordomo-announce" message and the list owner will receive a standard "SUBSCRIBE" notification, as shown above.

Such an "approve" cycle takes place if a user attempts to subscribe or unsubscribe any address that doesn't match the one in the header of their message, or if a user asks to subscribe to a "closed" list.

To find out who is on the majordomo-users list, Jane would send the following request:

```
From: Jane@Somewhere.ORG
To: Majordomo@GreatCircle.COM

who majordomo-users
```

and would receive the following response:

```
From: Majordomo@GreatCircle.COM
To: Jane@Somewhere.ORG
Subject: Majordomo results

>>>> who majordomo-users
Members of list 'majordomo-users':

brent@GreatCircle.COM (Brent Chapman)
Jane@Somewhere.ORG
Joe User <Joe@Elsewhere.GOV>
...
```

To find out which of the lists she's on that are served by a given Majordomo server, Jane would send the following request:

```
From: Jane@Somewhere.ORG
To: Majordomo@GreatCircle.COM

which
```

Majordomo would respond with:

```
From: Majordomo@GreatCircle.COM
To: Jane@Somewhere.ORG
Subject: Majordomo results

>>>> which
The address 'Jane@Somewhere.ORG' is
on the following lists served by
Majordomo@GreatCircle.COM:

        majordomo-users
```

To unsubscribe herself from the majordomo-users list, Jane would send a request such as:

```
From: Jane@Somewhere.ORG
To: Majordomo@GreatCircle.COM

unsubscribe majordomo-users \
        Jane@Somewhere.ORG
```

To which Majordomo would respond:

```
From: Majordomo@GreatCircle.COM
To: Jane@Somewhere.ORG
Subject: Majordomo results

>>>> unsubscribe majordomo-users \
        Jane@Somewhere.ORG
Succeeded.
```

The following message would also be sent to the list owner:

```
From: Majordomo@GreatCircle.COM
To: majordomo-users-approval@GreatCircle.COM
Subject: UNSUBSCRIBE majordomo-users

Jane@Somewhere.ORG has unsubscribed
from majordomo-users.
No action is required on your part.
```

If Jane's mailer automatically appended a signature to the end of all her outgoing messages, she could issue the "end" command as the last command of her messages to cause Majordomo to stop processing at that point. In addition, she could include blank lines or comments (anything following a '#' on a line is a comment, and is discarded before the line is processed) if she wanted to.

If the owner of the "majordomo-users" list wished to change the information file that is sent in response to "info" and "subscribe" requests, he could do that with a message such as:

```
To: Majordomo@GreatCircle.COM

newinfo majordomo-users PASSWORD
This is a revised information file
for the majordomo-users mailing list.
END
```

If the password used was the correct password for the list, Majordomo would replace the existing info file with the contents of the message to the "END"

marker (or the end of the message, if there was no marker). A wise list owner would probably include an "info majordomo-users" command after the "END" marker so that he could verify that the information update succeeded.

A list owner could also use a message like this to change the password for their list:

```
To: Majordomo@GreatCircle.COM

passwd majordomo-users OLD NEW
```

If the old password for majordomo-users was "OLD", then Majordomo would change the password to "NEW". For all Majordomo list owner operations that require passwords, knowledge of the password for the list is the sole authentication performed on the command. As I've said elsewhere in this paper, this isn't intended to be highly secure; it's merely intended to keep obnoxious people from making a nuisance of themselves by abusing list owner commands.

Note that Majordomo does not yet support continuation lines (a command line that ends with a backslash, indicating that the command continues on the next line) as shown above, though it is high on the list of features to be added. Continuation lines were used here for typesetting reasons.

Experiences with Majordomo

Majordomo is currently used to run the 17 SAGE mailing lists on USENIX.ORG, and to run the "Majordomo-Users" and "Majordomo-Announce" mailing lists at GreatCircle.COM (see the "Availability" section for more information about these lists). It's been in operation on USENIX.ORG since late June, 1992. In the two months between then and the time this paper was written, it has processed almost 1800 requests, all without encountering any major bugs or problems (though a number of minor bugs have been found and corrected). A number of other sites requested and received beta-test versions of the program, but I haven't heard back from any of them that they've begun using the software yet.

While Majordomo is similar to and inspired by Listserv, I haven't really attempted to make it a Listserv clone. I've chosen to use many of the same commands as Listserv, but I've often used slightly different syntaxes for some commands; for instance, the Listserv syntax for "subscribe" is "subscribe *list real_name*", as opposed to the Majordomo syntax of "subscribe *list* [*address*]". This may not have been a good idea; perhaps I should have either made the Majordomo syntax identical to the Listserv syntax or made it completely different. The copy of Majordomo running on USENIX.ORG uses the email address "Listserv", not "Majordomo"; it's not clear if that was a good idea, since it's not really Listserv.

Future Work

The next major set of features I intend to add are to support email retrieval of files through Majordomo. I need to look at mechanisms and syntaxes for making files and directories readable, writable, and searchable via email. I intend to support the notion of "open" and "closed" file directories (similar to the "open" and "closed" mailing lists currently implemented); only authorized people (where authorization might be determined by knowledge of an appropriate password, or by membership on a mailing list associated with the directory) will be able to retrieve files from "closed" directories. I also intend to support "writable" and "read-only" directories and files. I'm going to consider special support specificly for mailing list archives, to allow users to request only messages matching certain patterns or containing specified keywords from a given archive, rather than forcing them to retrieve the whole archive and do the search themselves.

At some point, I (or someone else) should go back in and make Majordomo work under "taintperl", so that the "wrapper" program won't be necessary. I firmly believe that "taintperl" is good and valuable, and that operating under it would improve the security of Majordomo; I just didn't have the time to work out all the details during my initial implementation phase.

I'd like to add a number of minor features to the program, including suppression of duplicate addresses in mailing lists (but is "joe@foobar.com" the same as "joe@workstation.foobar.com"?), recognition of unambiguous command abbreviations, support for continuation lines (some mailers insist on auto-wrapping text to fit an 80-column display; while this is often preferable to paragraph-long lines in text messages, it wreaks havoc with long Majordomo commands), support for a command indicating what return address Majordomo should use for its replies (for use by folks whose mailers generate broken reply addresses in the headers; this might, however, have security implications that would need to be carefully considered), and support for commands in the "Subject:" line of the message. I might look at making Majordomo more Listserv-compatible.

Availability

The package is available for anonymous FTP on machine FTP.GreatCircle.COM, in file "pub/majordomo.tar.Z". If you do not have anonymous FTP access, contact me (contact information is in the "Author Information" section, below), and I'll try to get a copy to you by email or some other means.

If you install Majordomo, please add yourself to the mailing list Majordomo-Users@GreatCircle.COM, which is for discussions concerning use of, problems with, and enhancements

for Majordomo. Announcements of new releases of Majordomo will be sent to Majordomo-Announce@GreatCircle.COM. You can add yourself to either or both lists by sending appropriate Majordomo commands to the electronic mail alias Majordomo@GreatCircle.COM.

Author Information

Brent Chapman is a consultant in the San Francisco Bay Area, specializing in the configuration, operation, and networking of UNIX systems. He is also currently Postmaster for SAGE (the USENIX Special Technical Group focusing on system administration issues). During the last several years, he has been an operations manager for a financial services company, a world-renowned corporate research lab, a software engineering company, and a hardware engineering company. He holds a Bachelor of Science degree in Electrical Engineering and Computer Science from the University of California, Berkeley. He can be contacted by electronic mail to Brent@GreatCircle.COM, by phone at +1 415 962 0841, by FAX at +1 415 962 0842, or by U.S. Mail to Great Circle Associates, 1057 West Dana St., Mountain View, CA 94041.

SysView: A User-friendly Environment for Administration of Distributed UNIX Systems

Philippe Coq & Sylvie Jean – Bull S.A. France

ABSTRACT

SysView is a tool for the administration of a group of heteregeneous UNIX systems interconnected over a LAN. The main objectives of this prototype are to give a unified view of heteregeneous systems, reduce the complexity of common administration in a distributed environment and provide a user friendly interface. The architecture of SysView is based on OSI Model for distributed systems management. The application part of this tool is written in a Prolog Language which contains object and class notions. This paper presents the functionalities and the architecture of this tool, as well as the application development environment we use to implement it.

Introduction

Nowadays, open administration of systems is a popular topic. Solutions are raising in in order to manage environment composed of interconnected heterogeneous systems (mainframes, PC, Unix). These solutions focus on the Enterprise Management and provide hypervision functionalities which allow the monitoring of these machines (alarm management, performance management). However, when they want to manage, i.e. to act upon machine resources, system administrators can only use the local specific management tool on the target machine. More and more, UNIX system administration means taking care of multiple computers, often issued from more than one manufacturer. We have developed a prototype of a tool whose objectives were to make administration tasks in a distributed environment easier.

In the next section we will describe the SysView solution from the user's point of view. Then we show the choosen architecture and later describe the object oriented application development environment. Last, we discuss the strength and the weakness of our approach.

Sysview Solution: The User's Point Of View

SysView has been designed to ease the UNIX administration of BULL machines on a local network (TCP/IP – Internet protocol). Figure 1 shows an example of a configuration in which SysView is administering two BOS2 systems and one BOSF system[1]. SysView provides the system administrator with an overall and unified view of the elements of the administered domain and their state: machines not accessible, terminal locked, printer disabled, file system full, etc.

SysView ensures consistent and efficient management of the administered machines. It avoids the administrator spending time in repeating same operations on every managed machine. This is done by offering some "multi-machines" operations. For example, it takes only one operation for creating the same user account on several machines or for sharing the same printer between several machines. This kind of operations favours better resource sharing of the domain: the same printer can easily be made available on several machines.

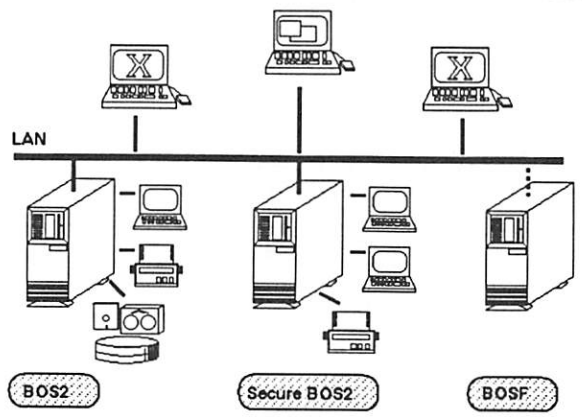


Figure 1: Example of a managed configuration

SysView is easy to use granted by its object oriented graphical interface and by many help on line facilities.

Sysview Features

Integration of system administration services

SysView is an environment for the administration of distributed UNIX systems. For the user, the integration of system administration services in

[1] BOS2 is the Bull operating system derived from UNIX AT&T System 5 Release 3.1. It exits a secure version of BOS2. BOSF is a system derived from OSF/1.

SysView results in:

- The pooling of system administration services. SysView's initial window displays all the available services and makes it possible to activate them (see Figure 2);

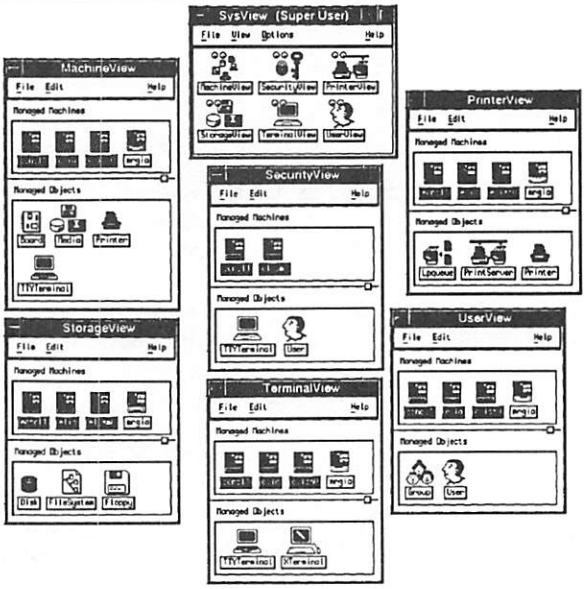


Figure 2: Initial panels of SysView applications

- The user interface consistency for the different services (format, dialogue). Each application provides the user with an homogeneous view of the resources of the administered domain and the means for managing them. The end user sees the administered domain as a set of machines and managed resources (represented graphically by icons). The operational mode is the same for all the applications: selection of machines, selection of the resource to be managed, selection of the action to be carried out;
- The ability of activating several applications simultaneously and in a coherent way. SysView manages the consistency of all the windows opened simultaneously, on the administered domain, by one or more SysView applications; As an example, imagine a system administrator using SysView. On the screen,several windows are displayed. Amongst them, an iconic list of terminals he/she has to manage and a form displaying the characteristics of a particular terminal (identification, status, type, line setting ...). Now, on a secure system, a user attempting many faulty logins causes this terminal to get locked. Consequently the icon representing this terminal, in the iconic list, is modified dynamically and reappers with a lock. In the form, the lock status toggle button is now switched on.

Simplification of Common Administered Tasks

SysView reduces the complexity of common Unix administration tasks in a distributed environment. SysView therefore makes a certain number of management choices. This comes down for implementing preferentially the procedures adopted by a large majority of system administrators.

High level Graphical User Interface

SysView is a set of X/MOTIF applications offering a user-friendly and object-oriented interface, in accordance with the MOTIF style guide. A set of sophisticated graphic representations allows the administrator to view the managed resources and to interact with them.

SysView uses icons for an overall representation of the administered resources. The icon depicts both the resource and its state (locked terminal , full file-system, disabled printer, ...). A particular iconic representation (framed icon) shows the existence of the same object on several machines (user accounts, groups, printer queues...). A click on an icon displays a popup menu which indicates the possible operations on the resource.

SysView also uses tables and forms for a detailed representation of the domain resources. Forms are used to display and update the attributes of the objects. Tables are used to display the attributes of objects distributed on several machines.

Assistance to the System Administrator

SysView is designed as an assistant to the system administrator, making available all the necessary information for carrying out the operation at hand.

Context sensitive menus associated to the icons, indicate the possible operations, at this time, on the managed resources. For instance, the unlock item appears in the menu associated to a user or terminal, only if these resources are locked. With on line Helps, the user can obtain an explanation for every input parameter of an administered operation. Mandatory parameters are easily identifiable.

SysView provides default values or choices of possible values to help the administrator in supplying the needed information. These values take the current state of the managed system into account. As an example, when creating a user on several machines, SysView provides the list of groups which are common across the different systems.

Internationalisation

Labels, help and error messages in SysView applications are internationalised. At present SysView supports both French and English. The user can switch between languages, at will, during a SysView session.

Alarm reporting

When an event requires human interaction, the administrator is warned by a change in the state of

the icon representing the resource in question. The events taken into account by SysView applications are:

- File system full;
- Swap overflow;
- Printer disabled by the spooler;
- User account or terminal locked due to too many faulty login attempts (secure systems).

Sysview Services

SysView is an integrated set of Unix system administration applications. Each application deals with the management of a particular type of resource.

- **StorageView** manages physical and logical storage. It facilitates formatting and partitioning operations and simplifies the operations to be carried out for managing a group of file systems in a multi-machine environment. In a single operation, StorageView distributes a file system on some or all the machines in the domain;
- **PrinterView** allows the system administrator to manage printer resources (print servers, printers, printer queues). As an example, PrinterView offers a "multi machine" operation for creating a printer queue on several machines and for exporting an existing queue on all the machines of the domain;
- **TerminalView** deals with the management of X terminals and asynchronous terminals (for more details see the example hereafter);
- **UserView** simplifies the administration of users accounts and groups and allows to standardize them easily on a set of machines (more details are given in the example, hereafter);
- **MachineView** offers the possibility to make an inventory of the hardware of the administered machines. The system administrator can list and display the characteristics of peripherals, boards and communication lines;
- **SecurityView** allows the Security Officer to easily manage some C2 security features on BOS2 secure systems. Main services are the configuration of system, user and terminal security parameters.

Several SysView applications can be activated in parallel. SysView manages the consistency of all the windows opened simultaneously on the administered domain. The top level view of each application displays all the machines in the domain and the classes of the administered objects as it is shown in Figure 2. The administrator first selects the machine(s) on which he/she wants to operate, the type of resources to be managed and then the operation to be performed.

Sysview Sessions: Examples

The examples below illustrate both SysView main features and services. X terminals management with terminalView is choosen to illustrate the simplification of administration tasks offered in SysView. Creating user accounts with UserView is an example of a "multi-machines" operation.

X terminals management with terminalView

TerminalView makes it possible to:

- Simplify the X terminal configuration phase;
- Display the X terminals of the domain and their state;
- Take into account the event "terminal locked".

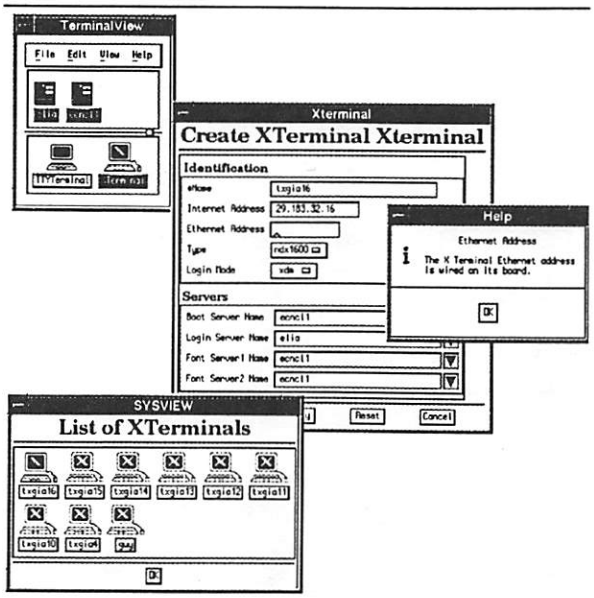


Figure 3: Installation of an X terminal with terminalView

Installing an X terminal is known to be a complex operation. Using SysView, it is considerably simplified and is carried out entirely from the management station, as shown in Figure 3. Powering up is the only action that needs to be performed at the X terminal.

To facilitate the configuration phase, TerminalView analyses the state of the machines in the domain to determine automatically the majority of the configuration information. TerminalView, for example, provides the list of possible login, boot and font servers.

TerminalView automatically executes the operations required to take into account the new terminal. As soon as it is powered up it displays automatically a login window.

Locked terminals are easily identifiable: their icons have a lock. The Security Officer can use SecurityView to put them back into an operational state by selecting unlock item in the menu.

Managing user accounts with UserView

UserView is a user and group management application which makes it possible to:

- Declare new user accounts and new groups on a group of machines;
- Standardize user's logins on several machines;
- Display the list of user accounts and groups on all the selected machines.

When creating a new account, UserView suggests a list of primary groups and possible secondary groups. When an administrator wants to declare the same user on several machines, the groups proposed are those common to the different machines. The user can leave to the system the choice of parameters such as the user and group identifiers (uid, gid).

UserView provides for the easy identification of user accounts and groups that are present on different machines. Figure 4 shows that users having an account on several machines are represented with framed icons, and locked user with a special icon.

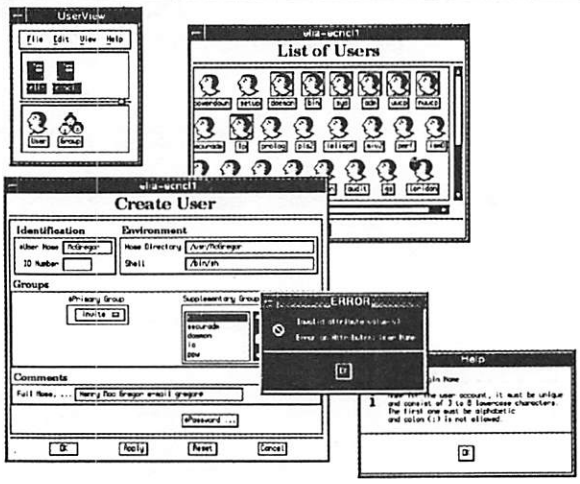


Figure 4: Creation of a user on all the machines of the domain

Global Architecture

ISO/OSI Management Model

The architecture of SysView is based on the ISO/OSI Model for Distributed Systems Management. In this object oriented model, the real resources of a distributed system are represenred by a set of managed objects.

A managed object is defined by its attributes, the operations which may be applied to it and the notifications it is able to emit. A managed object class defines a type of managed object with similar attributes, operations and notifications. A member of a managed object class is called a managed object instance.

Managed object classes are arranged in an inheritance tree.

The set of managed object instances of a distributed system makes up the Managed Information Base (MIB). Managed object instances have containment relationships defining a tree hierarchy called the containment tree.

Managed object classes are registered in a schema in which they are described using GDMO templates. These templates are based on Abstract Syntax Notation One (ASN.1).

General Architecture

In Figure 5 we see the three components of the architecture: applications, router, agents.

An operating system specific agent runs on each administered machine, the router and applications are processes running on one machine of the administered domain.

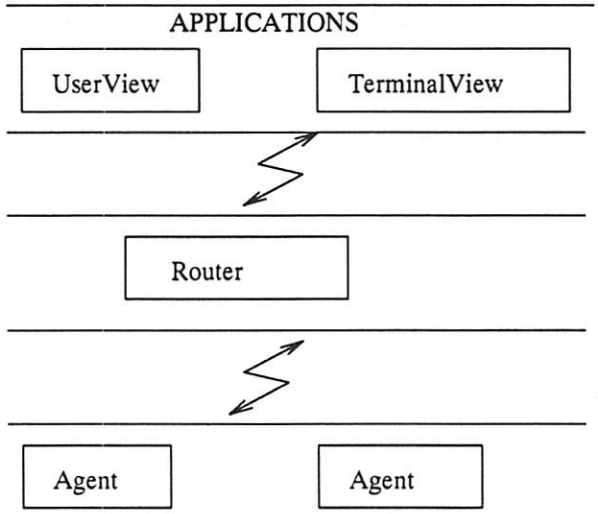


Figure 5: Global Architecture

Role of Components

Here are some of the components' roles:

Applications: the services offered by SysView are implemented by a set of management applications. An application displays the values of the managed object instances and handles the user interaction via the graphical user interface based on OSF/Motif. It interprets the user dialogue and generates requests which are sent to the router. The user can modify attribute values, create or delete instances, and perform other operations defined in the schema.

Router: It receives user's requests and dispatches them to the right agents. It gathers all the asynchronous responses from agents and sends them back to the application. It collects event notifications emitted by agents and transmits them to the application. Only few notifications are treated by the applications: File system full, Printer disabled, and the like.

The router also manages the schema of the MIB. Therefore applications can get information about managed object classes like the list of attributes of an object or the list of predefined operations associated to a class of objects.

Agents: An agent is a daemon running on an administered machine. Its role is to perform the correct operation on the real resources of a system. It executes requests that arrived from the router and sends it back responses and event notifications.

An agent has two parts : a generic part, the engine and specific parts named methods, specialized to particular classes of managed objects.

The engine receives requests, finds the concerned object (scope and filters), and determines the suitable method to activate.

The methods manage real resources as printers, users,... They are responsible for giving a view of those resources as managed objects. On a request such as GET or SET, those methods collect or update the information in the system.

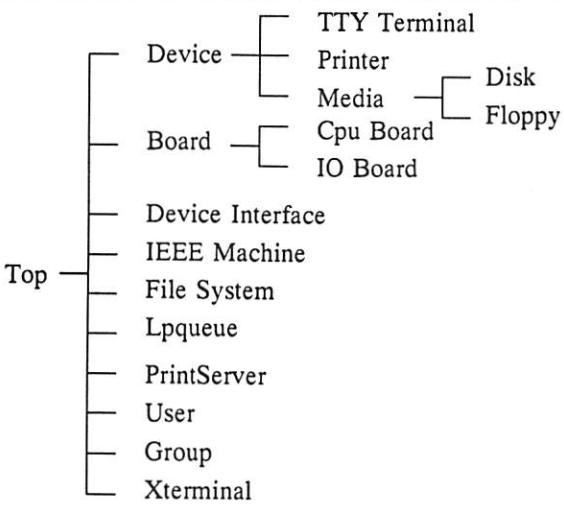


Figure 6: Inheritance Tree

Communication Protocol

We have analysed SNMP (Simple Network Mangement Protocol) and CMIS/CMIP (Common Management Information Service/Protocol) for achieving the communications Applications ↔ Router and Router ↔ Agents.

We have choosen CMIS because the protocol SNMP was too simple and the number of general methods was too small (get, set but no create or delete), to be applied in system management.

We have defined an interface based on CMIS and choosen to use a transport tool available on almost all UNIX systems and reliable : RPC (Remote Procedure Call).

Object Model

Our Object Model is very close to OSI Model. It includes:

- Classes
- Inheritance Tree
- Attributes
- Containment Tree
- CMIP verbs: Get, Set, Create, Delete, Action, Notification
- Scoping, filtering

Classes of Objects are described using GDMO (Guide for the Definition of Managed Objects). These classes are compiled and the corresponding schema is stored in the router and accessible at runtime by specific functions. Figure 6 shows the classes we defined and their inheritance relationship:

Application Development

The design of the applicative part of SysView was submitted to the following constraints:

- Architecture based on the OSI model for distributed system management;
- User interface in accordance with the OSF/Motif Style Guide;
- Easy integration of new services.

In order to abide by these constraints and to reduce development delays and costs, MIBIF, a development infrastructure for interactive administering applications was designed.

MIBIF stands for Managed Information Base Interface Facilities. It offers the programmer facilities for accessing MIB objects (data base side) and ready-made graphic representations for these objects (user-interface side). In addition, MIBIF ensures the coherency of all views opened on MIB objects by one or more SysView applications.

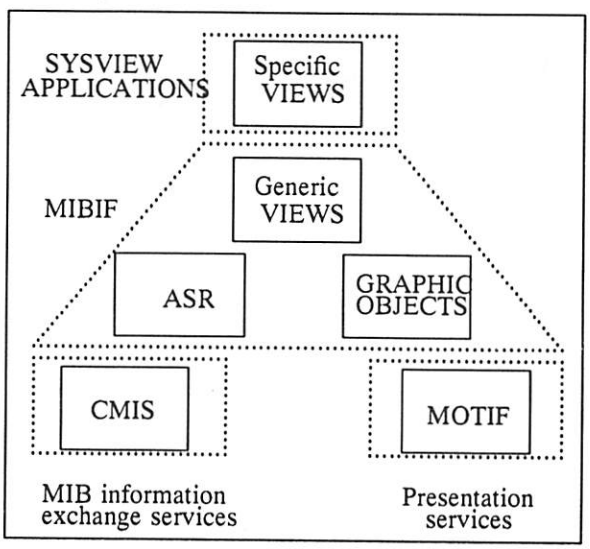


Figure 7: MIBIF architecture

MIBIF is relying on the use of SP-Prolog language. SP-Prolog is developed by Bull S.A company and integrates object oriented notions like classes, instances and messages (SmallTalk-like semantic)

MIBIF is made of the following components: Generic Views, ASR and Graphic Objects (See Figure 7). Generic Views are ready-made MIB object editors. They use the MIB information exchange services provided by the Abstract State Representation (ASR) and the presentation services offered by the Graphic Objects. Writing a new SysView application mainly consists of specializing Generic Views for specific editing purpose.

Abstract State Representation (ASR)

The ASR is a set of SP-Prolog predicates and classes, offering a convenient editing oriented interface to the MIB. It includes:

- High level ASR requests hiding the complexity of accessing the MIB via basic CMIS requests, of encoding/decoding ASN1.1 values, of getting information from the Schema,
- A mechanism for maintaining coherent multiple graphic representations of MIB objects also called views.

The subset of MIB managed objects manipulated by SysView applications, at a given time, is represented by ASR object instances. On these abstract representations of MIB objects are connected concrete graphic representations: the views. ASR instances are created and updated as a consequence of ASR requests generated by the end-users interactions on views. When these requests have been effectively performed on the corresponding MIB object by the agents, the ASR objects automatically send a notification to each connected view (modified attribute, deleted object..). The views which understand this notification are able to take into account the change and may update the screen. ASR notifications are also sent to views in response to a subset of CMIS notifications (alarms) received from the router. This can be summarized in Figure 8.

Graphic Objects

Graphic Objects are SP-Prolog classes built upon Motif Widgets. They make it easy to design the user interface of SysView applications.

Forms are used to display/update the attributes of managed objects and to prompt the parameters of an action. A macro language based on OSF/UIL allows a concise description of the form and simplifies the positionning of the widgets, their alignment, color and font.

Type editors are provided for the subset of ASN.1 types used in the Schema. Each type editor has several possible mapping on Motif Widgets. As an example, an enumerated type may be mapped on a TextEditor, a RadioBox or an option menu. Types editors do some syntaxic checkings (like checking of bounds for an IntegerValueRange)

Mechanisms are provided to support natives languages and to bound dynamic help windows on attribute or parameter labels in a form.

Lists of selectable objects are essentially used to display lists of managed objects. Textual or iconic representations may be chosen during a session by the end-user. A popup menu can be linked to an icon in order to display the available actions on the object selected. Different kind of interactions are provided (single select, multiple select, etc..).

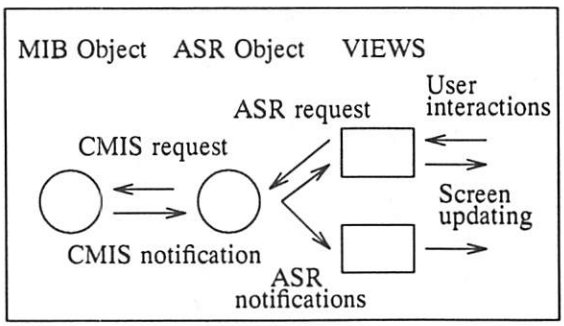


Figure 8: From end user to MIB: the flow of information

Views

Views are generic MIB Object editors. They are implemented as a set of SP-Object classes, each class being dedicated to an elementary editing operation on ASR objects. Main classes are: mibObjectList, mibObjectDisplayer, mibObjectCreator, mibObjectUpdater, mibObjectActionner. View classes use both ASR and graphical objects services.

For example, the standard behaviour of an mibObjectUpdater is to get the attributes of the managed Object, to present them in the correct form, and when the end user clicks on OK button to set the modified attributes of the managed Object.

User interactions are forwarded to views by the unique message handleEvent, all notifications from the MIB are sent to the views by the unique message asrNotification. Application developers may specialize a standard view by creating a subclass of the standard class and overloading the handleEvent or asrNotification methods.

Conclusions

We have developed this prototype for administering Bull UNIX systems. Complete agents were developed for DPX2/200 and DPX2/300 (multiprocessors) and for two BOS2 operating system versions (one secure and one not secure). Partial agents were also developed for BOSF operating system and for AIX.

From this experience we think the strong points are the following:

- The need exists for this kind of tools. We have tested our prototype on both Unix users and administrators. They have appreciated to have a uniform view of different systems. The simplification of complex tasks, especially those that involve more than one machine was felt as the strongest point.
- The ease of use and the training which is straightforward;
- New platforms or new systems can be integrated easily: the estimated cost for writing a new agent is 3 P/M (Person/Month);
- The use of object oriented language at the application level has permitted intensive reuse of the code and the MIBIF infrastructure has reduced strongly the development cost of applications;
- Performances are quite decent;
- "Multi-machines" tasks are time-saving in comparison with the classical machine per machine tasks;
- The use of this kind of tools avoids manipulation errors and assures a better coherence on the administered domain.

On the other hand, some problems still remain partially resolved in this prototype: error treatment and error recovery, the undo action that we do not integrate, and the semantic of some multi-machine are not easy to define (update of an object present on several machines with not identical attributes).

Finally we want focus on two types of problems that we have underestimated:

- The complexity of testing due to the lack of automated tool for testing such interactive graphic user interface, and the complexity generated by the great number of possible configurations in a distributed environment.
- The administration of such a software with multiple components running on several machines which requires a tool for installing and for ensuring the coherence of all these components.

Acknowledgements

We would like to thank all the people involved in the design, implementation, integration and tests of SysView. Finally, this project was made possible due to the encouragement, guidance and management provided by Philippe Donz.

References

[1] ISO 7498-4: Information Processing Systems – Open Systems Interconnection – Basic Reference Model – Part 4: Management Framework.

[2] ISO 10040: Information Processing Systems – Open Systems Interconnection – Systems Mangement Overview.

[3] ISO 10165-1: Information Processing Systems – Open Systems Interconnection – Mangement Information Services – Structure of Mangement information: Management Information Model.

[4] ISO 10165-2: Information Processing Systems – Open Systems Interconnection – Mangement Information Services – Structure of Mangement information: Definition of Management information.

[5] ISO 10165-4: Information Processing Systems – Open Systems Interconnection – Mangement Information Services – Structure of Mangement information: Guidelines for the Definition of Manged Objects.

[6] ISO 9595: Information Processing Systems – Open Systems Interconnection – Common Management Information Service.

[7] ISO 9596-1: Information Processing Systems – Open Systems Interconnection – Common Management Information Protocol.

[8] OSI/Network Management Forum – Forum Architecture. Issue 1, January 1990.

[9] ISO 8824: Information Processing Systems – Open Systems interconnection – Specification of Abstract Syntax Notation One (ASN.1).

Author Information

Philippe Coq is Software engineer at Bull S.A. He has worked in programming language team on Pascal compiler before to join the A.I. team. In this team he has developed the Bull prolog compiler and participated to the design and implementation of object oriented features in this language. In Sysview project, he was in charge of the definition of the application development environment. Mail: 1 Rue de Provence B.P.208 38432 Echirolles (FRANCE) Email: P.Coq@frec.bull.fr.

Sylvie Jean is Software engineer at Bull S.A. She has worked on implementation of Lisp language on Bull mainframes. In the A.I. team she has designed and developed several applications using Bull Prolog compiler and OSF/Motif. In the Sysview project she was in charge of the SecurityView application. Mail: 1 Rue de Provence B.P.208 38432 Echirolles (FRANCE) Email: S.Jean@frec.bull.fr.

Depot: A Tool for Managing Software Environments

Wallace Colyer & Walter Wong – Carnegie Mellon University

ABSTRACT

`Depot` is a software management tool which provides a simple, yet flexible, mechanism for maintaining third party and locally developed software in large heterogeneous computing environments. `Depot` integrates separately maintained software packages, known as collections, into a common directory hierarchy consisting of a union of all the collections. This common directory is defined as the software environment. A set of configuration options manages interactions and intersections between collections in the environment.

`Depot` facilitates the introduction, update, and removal of collections in a software environment. Custom environments and complete test environments can be easily created for individual machines or for sets of machines. Collections with unexpected problems can either be replaced with previous versions or removed. Individual collections or files can be moved from remote filesystems to the local disks of workstations without the concern that those files may become outdated. All this is achieved with minimal wasted disk space and administrative overhead.

Introduction

The installation and maintenance of application software on UNIX platforms has traditionally been a difficult and time consuming process. Many difficulties result from inadequate software release and environment control tools. The situation is aggravated by a complete lack of industry standards, the common use of hard coded paths for file dependencies, and unreasonable assumptions that many software providers make of the installation environment.

The emergence and popularity of distributed computing has compounded the management problem. A large heterogeneous environment, with thousands of workstations and hundreds of software packages, aggravates the existing problems and adds new ones which must be overcome.

To properly manage a software environment several issues must be resolved. An inventory must be maintained containing the origin of all the components of the environment. Software must be thoroughly tested independently, as well as in the destination environments. If a critical problem escapes the testing process, the software environment must be smoothly restorable to a previous working state.

In a distributed software environment, there is a need to distribute and install software on remote machines with different architectures, customizations and configurations. The procedures required must minimize the workload of the system administrators.

Many solutions that manage a distributed software environment often bring back load and availability problems of timesharing systems by increasing the dependence on centrally maintained services. To prevent this, workstations should be able to locally cache commonly used files, as well as maintain a core set of functionality in case of server or network failures.

A software release management system in complex environments should handle the following issues:

- Distribution
- Installation
- Customization
- Testing
- Removal and restoration

By segregating the environment into discrete manageable objects, it is possible to address all of these issues. These objects can then be layered to create the user visible environment. Thus, the environment can be looked at as either a whole or in parts.

Motivation

In the past, the software maintainers of the Andrew system installed software, following the general UNIX philosophy, directly into the **/usr/local** tree. As the number of applications multiplied, the maintenance process became increasingly difficult. For example, because no records were kept of the files that were installed with a software package, often outdated files were left in the system, wasting valuable disk space.

Another problem occurred when two applications had files with the same name installed in the same directory; the conflicting file would be overwritten during the installation process. This would lead to all sorts of subtle problems, especially

if the files were significantly different between the two applications.

When our software manager began posting lists of hundreds of files and asking if anyone knew whether they still belonged in **/usr/local**, it became apparent that we had a serious problem. We identified four key components necessary to solving this problem: independence, integration, mobility and simplicity.

Independence

The problem of keeping track of software can be solved by separating sets of related software into independent directory hierarchies called *collections*. The collection abstraction reduces the complexity of the software environment by creating smaller, well defined, working groups. Each collection is kept in separate locations, so it is simple to determine the origin of its files. This facilitates finding and reporting problems, as well as cleaning up the environment when updates occur or when the software package becomes obsolete. Furthermore, a complete software package can be distributed or shared by simply specifying the path to the collection.

`The Depot` [Manh90], developed at the National Institute of Standards and Technology (NIST), splits applications into different collections; however, no integration is done. To access files in a collection, the user must be aware of this separation and have a very long list of items on his path. We considered this approach to be too cumbersome in the Andrew environment.

Many problems had been encountered when making changes to the search paths of over 10,000 users of our system, many of whom access our filesystem from departmental computing facilities where we do not have administrative access. These departments wish to import a single directory hierarchy to their machines, such as **/usr/local**, in order to use our software. It is convenient for there to be a single path to all software in that hierarchy, i.e., **/usr/local/bin**. While it is possible to have configuration files which all users access to set the paths, it is difficult to keep those files up to date and to ensure users in other departments will use these centrally maintained site configuration files.

Integration

Not only is it necessary to maintain independent collections for the sake of administrative convenience, it is also important to unify the collections into a single directory hierarchy. This helps the users understand the environment as they do not have to look in many different places for system software. This provides, as well, a way for applications to share common directories.

Many applications need to share directories where common files are kept. Index files are often kept that must be updated whenever any files change in these directories. Two common examples of this are: `X11` fonts and `man` pages. With total separation, it becomes increasingly difficult to seamlessly integrate the environment.

Maintaining independent collections does not, however, address the problem of two applications installing binaries with the same name – path conflicts and ordering problems still remain.

These problems can be addressed by integrating the independent collections into a common directory hierarchy and forcing conflict resolution.

Mobility

The most obvious reason for wanting to efficiently move software in and out of environments is for testing. If environments can be created without regard to the actual location of the collections, software can be tested by creating a duplicate destination environment. It should be possible to generate a software environment that is identical to the current released environment, with the exception of the collection to be tested. If major problems were uncovered, it should be simple to restore the old environment quickly.

Distributed filesystems illustrate the concept of mobility. Collections should be able to move between the remote filesystem and the local disk of the client workstation. Collections on the local disk should be updated when new versions come out as transparently as possible. Rarely used applications may be stored on the remote filesystem, thus conserving local disk space. Commonly used or important applications may be stored on the local disk, thereby increasing access speed and availability. Regardless, the actual location of the software should be transparent to the end user.

While other systems, such as `Xhier` [Sell91], have recognized the need for independence and integration, no package that we examined addressed the issue of mobility. Most provide only one environment, for example, `Xhier`'s **/software** and NIST's `Depot`'s **/depot**. Ideally, multiple environments would be possible. We have found it desirable to have an environment for fully supported software, **/usr/local**, and another for "unsupported but useful" software, **/usr/contributed**. Moreover, it should be possible to easily move collections from **/usr/contributed** to **/usr/local** and vice versa.

Simplicity

Paul Anderson [Ande91] described a method of tracking software by tagging files in each collection with a unique UNIX userid (uid). This approach satisfied the independence, integration, and, to an extent, the mobility requirements. However, the process has a good deal of complexity, since developers are required to do extra work to utilize the system. Additional tools are required to track and maintain the collections. Furthermore, in a decentralized distributed environment, password files must remain homogeneous across all machines.

Alternatively, simplicity and understandability could be maintained by having each collection imported into the software environment by using its own directory hierarchy. For example, a file placed in the **bin** directory of a collection should appear in the **bin** directory of the environment. The software installer should only need to determine the desired directory hierarchy. When the collection appears in the environment it will reflect that hierarchy. The environment maintainer should only need to decide which collections to integrate into the environment and how to resolve any conflicts, or when two collections try to install the same file in the same location.

Additionally, the system can be kept simple by not incorporating the distribution mechanism into the program. For example, a distributed filesystem, such as `AFS`[1] [Saty85], or standard software distribution tools such as `rdist` and `SUP` [Shaf88] may be used. Distribution may become important for certain classes of machines, such as laptops, and other computers connected by slow or unreliable network connections, but any distribution solution should still be external.

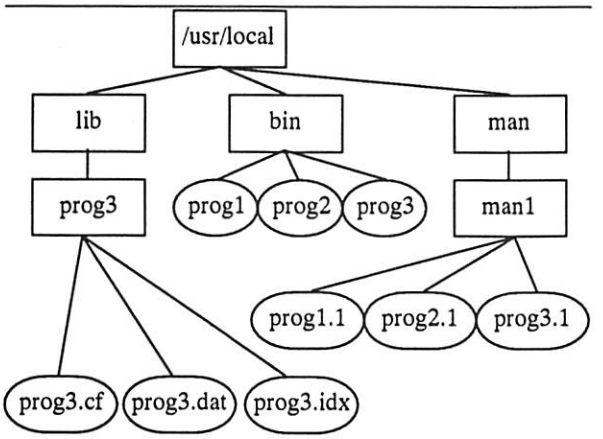


Figure 1: Simple **/usr/local** Environment

Implementation

`Depot` creates a system which requires relatively little work to setup and maintain multiple environments, to easily allow software to be quickly installed and backed out, and to allow individual workstations to be customized. This is done in a manner that minimizes the overall complexity of maintaining large software environments. `Depot` achieves the goals of independence, integration, mobility and simplicity. The system is implemented by integrating multiple independent collections into a single directory hierarchy and allowing specific customizations via configuration files.

[1]The Andrew Filesystem (`AFS`) is a scalable distributed filesystem available from the Transarc Corporation

With each invocation, `depot` processes a single software environment. The software environment starts with a specified directory hierarchy and encompasses everything within it, including subdirectories. Figure 1 shows a simple **/usr/local** environment.

`Depot` defines the environment as the union of a set of software collections. Figure 2 shows a way collections can be stored in the `depot` framework. In the environment, the **depot** directory is special. This directory stores the database and configuration files and, in this case, stores the collections.

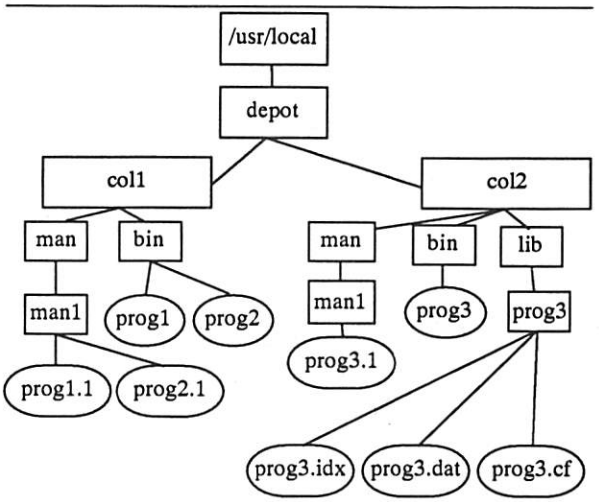


Figure 2: Collections

The environment is customized through a set of configuration options. These options determine which collections will be integrated into the environment and how they will be integrated.

Collections can be integrated into an environment in several ways:

- By listing specific collections and the paths to their location
- By providing search paths where the first instance of each collection within the path will be used
- By placing the collections in the **depot** directory of the environment, as shown in figure 2
- Or by using a combination of these methods

Before integrating the collections into an environment, `depot` verifies that the environment is consistent. Files in the in the environment that do not belong to any collection or are not marked as special files are deleted. `Depot` will then check to see if any of the collections on its paths have been changed, added or deleted. All the new collections will be added to the environment, all removed collections will be deleted, and any necessary changes for modified collections will be made.

As `depot`'s last action, other binaries can be run if a specified collection changes. This addresses the issue where special files need to be generated.

For example, often multiple collections install files into a common directory. Index files are often kept of the contents of these directory, such as **fonts.dir** in the `X11` font directories. Future versions of `depot` will have configuration options to run the commands whenever the directory structure changes.

In the integration process, there is the chance that files, from different collections, will export to the same place. For example, if two collections both have the file **bin/foo**, then a *conflict* has occurred between the two collections. `Depot` will exit at this time. To resolve the conflict, the environment maintainer can specify that one collection is to override the other. Alternatively, the environment maintainer may request the collection maintainer to move files or directory hierarchies by changing the collections or by using collection specific configuration files.

There are two ways a collection can be integrated: copying or linking. For collections that are linked, symbolic links are made from the environment to their location in the collections specific directory. To reduce the overhead of the links, they are made at the directory level wherever possible. With the copy option, every file and directory is copied into the target environment.

In earlier versions, `depot` was only able to operate at the collection level. There was no way to copy or link individual files or directories; the entire collection was either copied or linked. This strict separation, with no single file operations, proved to be too restricting. Target specific options now permit individual files or directories to be copied, linked, deleted or ignored, regardless of the collection of origin. For example, several collections may install fonts into the **lib/X11/fonts** directory, and the environment maintainer may wish them to always be copied, regardless of the collection they came from. On the other hand, the environment maintainer may choose to link all the files integrated into the **man** or **doc** directories to conserve space. Since these options work only by changing the behavior when a file is mapped out of a collection into the target directory, and they do not modify the resulting structure, the sanctity of the collection is maintained, and a great deal of flexibility is achieved for the environment maintainer.

Figure 3 presents an environment integrated by using symbolic links. This environment is generated by `depot` from the collections in Figure 2 and would present the same structure, to the user, as the one shown in Figure 1. In this case, the environment was generated by symbolic links which are represented by the shaded objects. The arrows point to the actual location of the files or directories.

When creating an environment with symbolic links `depot` performs link optimization in order to link at the highest possible level of the collection hierarchy. This reduces the number of symbolic links `depot` must make. Figure 3 shows this process. Since the **lib** directory only exists in *col2*, `depot` links **/usr/local/lib** directly to the **lib** directory in the *col2* collection. If another collection later introduces a file into a directory which has been optimized at a higher level, the link will be

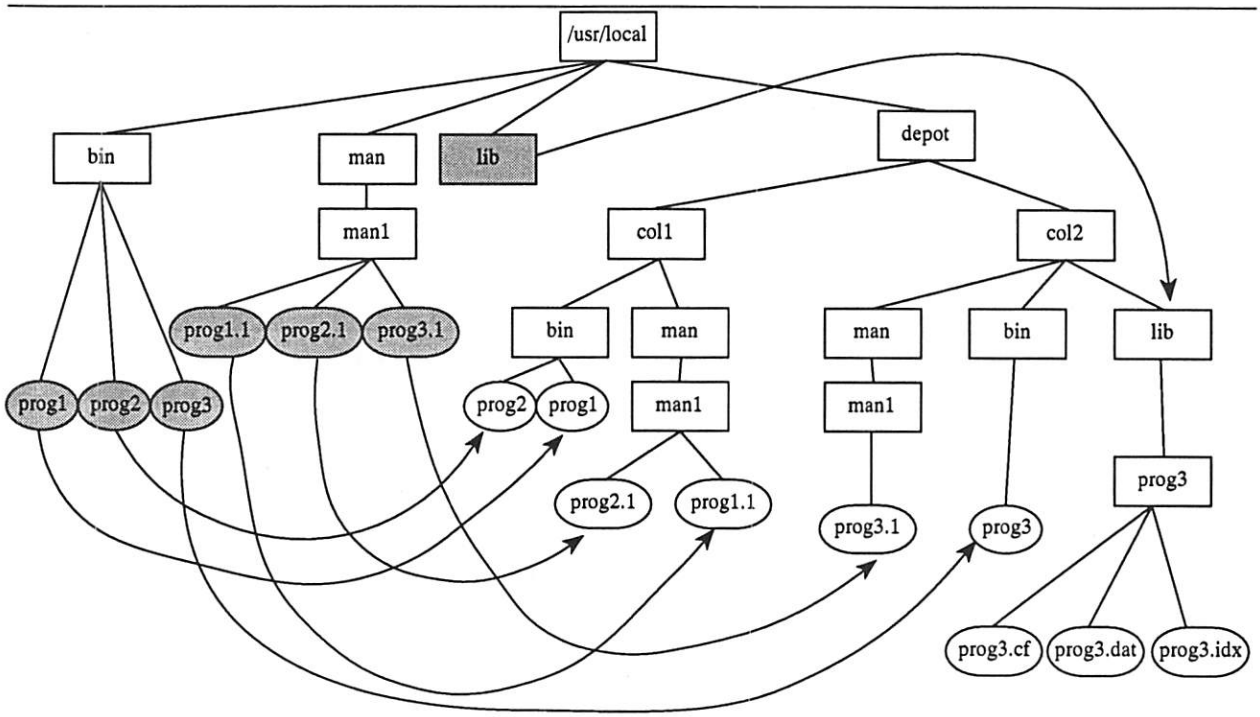


Figure 3: Symbolic Link Environment

removed and all the files at the lower level will be linked in. It will again attempt to link any subdirectories of the previously optimized directory new links are made. Many collections (e.g., the `Framemaker` publishing package), have hierarchies of thousands of files into which it is unlikely any other application would ever introduce files.

In our environment, the software environment is first integrated on a shared filesystem. Most workstations may just access the environment in that manner. However, workstations with more local storage may move the environment to their local disk. This creates the situation where the client should be updated whenever software is updated in the shared filesystem. For example, a symbolic link could exist from the local disk to a file in a collection on the shared filesystem. If, the collection maintainer changes the collection by removing the file, there will be the case where a symbolic link is pointing at a nonexistent file.

To resolve the consistency problems, the client workstation must either run `depot` immediately, or there must be a way for the local workstation environment to remain consistent and fully functional until a scheduled run of `depot` occurs. Requiring all participating machines to run `depot` simultaneously in a large workstation installation neither feasible nor practical. To allow workstations to run `depot` on their own time frame, we added the concept of `depot` version numbers. When a new version of a software package is released to the environment, it is mounted with a higher version number. The highest version is selected and integrated into the environment. A reasonable number of versions are kept so no collections will be erased before a workstation has an opportunity to run `depot`. Thus, functionality and consistency of the environment is preserved.

By integrating multiple independent collections into a single environment, `depot` achieves independence and integration. The search paths, version numbers, and different updating strategies provide mobility by allowing the integration of new or different versions of a software package from different locations. Finally, the mirroring of directory hierarchies and simple configuration options are easy for administrators and software developers to understand and use, thereby achieving the goal of simplicity.

Limitations

`Depot` only operates inside a single environment at one time. Software managed in **/usr/local** cannot be moved by `depot` outside of **/usr/local**. Software or files that need to be copied into the operating system areas will require another program to do so. There also lacks a mechanism to scan the entire environment for conflicts. This makes building the environment for the very first time a somewhat longer and more tedious task.

Currently, `depot` is somewhat inefficient at dealing with very large environments. The time to search its databases and to *stat*(2) source directories for changed collections increases undesirable as the environment grows. Some performance enhancements have been made by introducing code specific to `AFS` volumes[2] [Side86], but these have not been sufficient. A network server or hint files, containing modification dates of collections and information about their tree structure, may be needed. A complete rewrite of the database and customization handling routines is planned.

Some additional tools are required for distribution and for detailed tracking of software in the environments. Colyer, et. al., [Coly92] provides an overview of the tools used with `Depot` to manage the Andrew Software Environment. Mark Held [Held92] provides a more detailed explanation of the environment. Also, as a result of our `AFS` environment, the issue of architecture differences is not addressed by `depot`. This issue must be handled by the distribution system.

We are planning for `depot` to replace `package` [Youn85], our current host configuration tool, and make `depot` a workstation manager. The environment would be the operating system of the workstation. Each operating system release would be a collection where minor release levels would override the major release. Layered operating system products would also be collections in the environment. In addition to this, a hierarchy of overrides are also required where "Andrew" changes would override operating system defaults. Further, departmental changes would override "Andrew" changes and finally local workstation changes, with the highest priority, would override all other changes.

Conclusion

In the Andrew environment, where `depot` was developed, it would be unthinkable to return to the situation such a tool was available. Installations were lengthy, error prone processes. Often the installation of a new application would break previous applications. There was no smooth way to restore the environment to a previous state. Even though numerous man hours were put into maintaining the environment, the system was essentially in a state of anarchy.

Today, even with multiple environments, software can be easily installed and removed from the system. Individual workstations can be customized to achieve a degree of network independence with minimal effort by the central staff or workstations owners. Much of `depot`'s success can be attributed to the four factors discussed earlier:

[2]Volumes are containers of UNIX filesystems, similar to disk partitions. They are the administrative unit of AFS.

independence, integration, mobility and simplicity. The concept of combining independence and integration provided the manageability we needed without sacrificing the consistency that users demand. Mobility gives us flexibility in configuration and testing. Finally, the simplicity has made it popular with developers and allows us to integrate `depot` with other tools, rather than trying to make `depot` a "kitchen sink" tool. `Depot` has proved to be a flexible mechanism for maintaining our software environment.

Acknowledgments

Without the considerable work done by a large number of people `depot` would not have been possible. Many more people were influential in the creation of `depot` and the software management procedures along side it.

In 1988, `depot` was born during a set of software management brain storming sessions attended by Wallace Colyer, Mark Held, Ted McCabe, and David VanRyzin. Each member of this group contributed to the creation of `depot`. There were many useful insights gained from previous software management strategies developed in conjunction with the Information Technology Center (ITC) and groups within the Academic Services division at Carnegie Mellon University. Mike Accetta and other members of the School of Computer Science were very helpful during our initial consultations in explaining the strengths and weaknesses of their **/usr/misc** software management system. **/usr/misc** provided the initial ideas for the creation of `depot`. The original prototype was written in `perl` [Wall91] by Wallace Colyer in 1989. It has since been rewritten in C by was written by Sohan C. Ramakrishna-Pillai.

We would like to thank Terilyn Gillespie and Dawn Neuhart for helping to, once and for all, finish this paper.

A final set of thanks goes to Mark Held, who is leading the effort of maintaining our local and third party software. He also undertook the enormous project to migrate all our software into the new architecture and has produced an excellent software development environment based on the framework provided by `depot`.

Availability

`Depot` is available via anonymous ftp from *export.acs.cmu.edu* [128.2.35.66] in **/pub/depot**. `Depot` is also available via AFS in **/afs/andrew.cmu.edu/system/archive/cmu/depot**.

Any questions about `depot` can be sent to depot+@andrew.cmu.edu. `Depot` does not require `AFS`.

References

[Ande91] Anderson, Paul. "Managing Program Binaries In a Heterogeneous UNIX Network." *LISA V Proceedings.* 1991. pp. 1-9.

[Coly92] Colyer, Wallace; Held, Mark; Markley, David, and Wong, Walter. "Software Management in the Andrew System." *AFS User's Group Proceedings.* June 1992.

[Held92] Held, Mark, and Neuhart, Dawn. *Software Management in the Andrew Distributed* UNIX *System at CMU.* Computing Services, Carnegie Mellon University. 1992.

[Manh90] Manheimer, Kenneth, Warsaw, Barry, Clark Stephen, and Rowe, Walter. "The Depot: A Framework for Sharing Software Installation Across Organizational and UNIX Platform Boundaries." *LISA IV Proceedings.* 1990. pp. 37-46.

[Saty85] Satyanarayanan, M.; Howard, J. H; Nichols, D. A.; Sidebotham N., and Spector A. Z. "The ITC Distributed File System: Principles and Design." *Proceedings of the 10th ACM Symposium on Operating System Principles.* 1985.

[Sell91] Sellens, John. "Software Maintenance in a Campus Environment: The Xhier Approach." *LISA V Proceedings.* 1991. pp. 21-44.

[Shaf89] Shafer, Stephen, and Thompson, Mary. *The SUP Software Upgrade Protocol.* Carnegie Mellon University, School of Computer Science. 1988. Available from mach.cs.cmu.edu in **/usr/mach/public/doc/sup.ps**.

[Side86] Sidebotham, R. N. "Volumes: The Andrew File System Data Structuring Primitive." *Technical Report CMU-ITC-053.* Information Technology Center, Carnegie Mellon University. 1986.

[Wall91] Wall, Larry, and Schwartz, Randal L. *Programming perl.* O'Reilly and Associates, Inc. 1991.

[Youn85] Yount, Russell. *Package.* Academic Services. Carnegie Mellon University. 1985.

Author Information

Wallace Colyer is the Andrew Systems Manager at Carnegie Mellon University. He began as a User Consultant specializing in workstation administration issues. Time and a variety of departmental reorganizations found him in charge of the entire system. Send Email to wally+@cmu.edu.

Walter Wong obtained a B.S. in Cognitive Science at Carnegie Mellon University in 1991. By that time, however, he was already involved with system administration issues in a distributed computing environment. Rather than basking in the glory of a fine graduate school in a small college town, Walter stayed at Carnegie Mellon to be a system administrator and programmer for the Andrew Systems Group. Send Email to Walter.C.Wong@cmu.edu.

Both authors may be reached via the postal system at:

Computing Services
Carnegie Mellon University
4910 Forbes Avenue
Pittsburgh, PA 15213-3891

Examples

The following example is a simple use of `depot` to integrate two collections into an environment called **/usr/test**. A directory called **depot** is created under **/usr/test** which houses the configuration files and collections. There are two collections *col1* and *col2*. Each has its own directory hierarchy which is shown below.

```
/usr/test/depot/col1
      bin/prog1
      bin/prog2
      man/man1/prog1.1
      lib/libprog1.a

/usr/test/depot/col2
      bin/prog3
      man/man1/prog3.1
      lib/libprog3.a
```

A simple configuration file is created which tells `depot` to use the modification times to see if a file has changed.

```
% cat /usr/test/depot/custom.depot
usemodtimes: true
```

Running `depot` will integrate these two collection, *col1* and *col2,* with a common **man**, **lib**, and **bin** directory.

```
% cd /usr/test/depot
% depot -B [this builds the initial database]
% depot -va [-v makes depot verbose; -a updates all collections]
DIRECTORY ..
MKDIR ../man
MKDIR ../man/man1
LINK depot/col2/man/man1/prog3.1 ../man/man1/prog3.1
LINK depot/col1/man/man1/prog1.1 ../man/man1/prog1.1
MKDIR ../lib
LINK depot/col2/lib/libprog3.a ../lib/libprog3.a
LINK depot/col1/lib/libprog1.a ../lib/libprog1.a
MKDIR ../bin
LINK depot/col2/bin/prog3 ../bin/prog3
LINK depot/col1/bin/prog2 ../bin/prog2
LINK depot/col1/bin/prog1 ../bin/prog1
Backing up old database .. done
Moving in new database .. done
```

The following is the directory hierarchy, reflecting the union of *col1* and *col2*, created under **/usr/test**.

```
/usr/test
      bin
            prog1
            prog2
            prog3
      lib
            libprog1.a
            libprog3.a

      man/man1
            prog1.1
            prog3.1
```

By adding the mapcommand line to the configuration file, the actual files are copied out of the collection and into the **/usr/test** hierarchy.

```
% cat custom.depot
usemodtimes: true
```

```
*.mapcommand: copy
% depot -va
DIRECTORY ..
DIRECTORY ../man
DIRECTORY ../man/man1
REMOVE ../man/man1/prog3.1
COPY depot/col2/man/man1/prog3.1 ../man/man1/prog3.1.NEW
RENAME ../man/man1/prog3.1.NEW ../man/man1/prog3.1
UTIMES ../man/man1/prog3.1 Wed Aug 19 10:52:11 1992
REMOVE ../man/man1/prog1.1
COPY depot/col1/man/man1/prog1.1 ../man/man1/prog1.1.NEW
RENAME ../man/man1/prog1.1.NEW ../man/man1/prog1.1
UTIMES ../man/man1/prog1.1 Wed Aug 19 10:51:19 1992
DIRECTORY ../bin
REMOVE ../bin/prog3
COPY depot/col2/bin/prog3 ../bin/prog3.NEW
RENAME ../bin/prog3.NEW ../bin/prog3
UTIMES ../bin/prog3 Wed Aug 19 10:51:56 1992
REMOVE ../bin/prog1
COPY depot/col1/bin/prog1 ../bin/prog1.NEW
RENAME ../bin/prog1.NEW ../bin/prog1
UTIMES ../bin/prog1 Wed Aug 19 10:51:03 1992
REMOVE ../bin/prog2
COPY depot/col1/bin/prog2 ../bin/prog2.NEW
RENAME ../bin/prog2.NEW ../bin/prog2
UTIMES ../bin/prog2 Wed Aug 19 10:51:03 1992
DIRECTORY ../lib
REMOVE ../lib/libprog3.a
COPY depot/col2/lib/libprog3.a ../lib/libprog3.a.NEW
RENAME ../lib/libprog3.a.NEW ../lib/libprog3.a
UTIMES ../lib/libprog3.a Wed Aug 19 10:52:21 1992
REMOVE ../lib/libprog1.a
COPY depot/col1/lib/libprog1.a ../lib/libprog1.a.NEW
RENAME ../lib/libprog1.a.NEW ../lib/libprog1.a
UTIMES ../lib/libprog1.a Wed Aug 19 10:51:29 1992
Backing up old database .. done
Moving in new database .. done
```

If a new file is added to a collection, it will be integrated into the environment by running **depot** again. Thus, the file **/usr/test/depot/col2/bin/prog4** is added to the collection *col2*.

```
% depot -va
DIRECTORY ..
DIRECTORY ../bin
COPY depot/col2/bin/prog4 ../bin/prog4.NEW
RENAME ../bin/prog4.NEW ../bin/prog4
UTIMES ../bin/prog4 Wed Aug 19 10:59:10 1992
Backing up old database .. done
Moving in new database .. done
```

A new environment, **/usr/test2**, can be created that builds upon the collections in the **/usr/test** environment. Under the **depot** directory in **/usr/test2** we have a newer versions of *col2* and a new collection called *col3*.

```
/usr/test2/depot/col2
      bin/prog3
      bin/prog4
      man/man1/prog3.1
      lib/libprog3.a

/usr/test2/depot/col3
      bin/prog5
      lib/prog5/fonts/prog5.font
```

```
% cd /usr/test2/depot
% cat custom.depot
usemodtimes: true
*.searchpath: /usr/test2/depot,/usr/test/depot
% depot -B
% depot -va
DIRECTORY ..
MKDIR ../man
MKDIR ../man/man1
LINK /usr/test2/depot/col2/man/man1/prog3.1 ../man/man1/prog3.1
LINK /usr/test/depot/col1/man/man1/prog1.1 ../man/man1/prog1.1
MKDIR ../lib
LINK /usr/test2/depot/col2/lib/libprog3.a ../lib/libprog3.a
LINK /usr/test/depot/col1/lib/libprog1.a ../lib/libprog1.a
MKDIR ../bin
LINK /usr/test2/depot/col3/bin/prog5 ../bin/prog5
LINK /usr/test/depot/col1/bin/prog2 ../bin/prog2
LINK /usr/test/depot/col1/bin/prog1 ../bin/prog1
LINK /usr/test2/depot/col2/bin/prog4 ../bin/prog4
LINK /usr/test2/depot/col2/bin/prog3 ../bin/prog3
LINK /usr/test2/depot/lib/prog5 ../prog5
Backing up old database .. done
Moving in new database .. done
```

This creates a new environment, *test2,* by using *col1* from the *test* environment along with the newer version of *col2* and a new collection, *col3,* from the *test2* environment. For **lib/prog5**, link optimization was accomplished. Since the only collection installing into the **lib/prog5** directory was *col3* and the entire directory was being imported, the symbolic link was made at the highest point in the tree.

Software Distribution and Management in a Networked Environment

Ram R. Vangala & Michael J. Cripps – NCR
Raj G. Varadarajan – AT&T

ABSTRACT

Decentralization is an ever-increasing problem in today's networked environments. While the centralized computing paradigm has worked well for a long time, the de-centralized paradigm is still in its infancy. This paper discusses distribution and management of software in a highly networked environment. It discusses options, obstacles, a strategy, model, architecture, and implementation.

Introduction

During the past decade or so, de-centralization of computing resources in the corporate environment has been gaining considerable momentum. Corporations are seriously considering relegating their mainframes to more mundane and batch oriented tasks. More and more applications used by their employees on a daily basis are being moved onto microcomputers and workstations that are based on non-proprietary operating systems such as the UNIX System, OS/2 and DOS. Such a move, in the long run, is likely to remove the chances of single point of failure in a corporate computing structure and make computing resources more available, easier to access, and easier to use.

In spite of its drawbacks, the centralized computing paradigm has been perfected over a period of time and has matured into a very stable environment. Almost all the centralized proprietary systems have excellent tools to manage those environments. To name a few, these include tools to monitor performance of a system, to perform backups and restores, and to manage software.

The de-centralized paradigm is still in its early stages in terms of maturity. There is a serious lack of tools to manage these environments. Unless it is made as reliable as the centralized paradigm, users will not be able to really decentralize all their computing resources. When this level of maturity is achieved we will be able to really call it a distributed environment, instead of just a de-centralized environment.

From an operations perspective, at least the following areas have to be made far more reliable and easier use before distributed computing can become a reality for mission critical applications:

- Backup and Restore
- Software Distribution

This document explores the area of software distribution in a de-centralized environment. It discusses the various options available to distribute and install software on geographically diverse networks of heterogeneous systems. It also discusses the obstacles and practical difficulties involved in accomplishing this task. Based on the options and obstacles, a strategy to perform software distribution is presented. The strategy is then reflected in an architectural framework. A reference implementation of the framework and the future growth options of the implementation are also discussed. The reference implementation discussed in this document is a commercial product that is readily available as a part of the StarSENTRY suite from the NCR Corporation.

Options

Reliable distribution of software to systems in a heterogeneous environments may be accomplished using many different approaches depending on the number of systems on the network and their geographic spread. In this section, we discuss the various options available to us and their limitations.

Manual Distribution

Manual distribution of software by having an individual present at each system, may be a feasible solution if all of the following conditions are true:

- The number of systems on the network is small.
- Systems on the network are not too far apart from each other.
- The number of software packages installed on the systems is small.
- The frequency of updates to software is small enough that success or failure of new installations can be manually verified.

Ad Hoc Solutions

Ad Hoc solutions created by integrating various tools available in house may be a reasonable solution if all of the following conditions are true:

- Systems on the network are not too far apart from each other.
- There are little or no differences between systems in terms of software configuration.
- The number of software packages installed on the systems is small.

- Sophisticated enough tools are available to manipulate status information from installations to check for success/failure.

Electronic Software Distribution

The most flexible, reliable and comprehensive solution to the problem of software distribution is an electronic software distribution scheme. A software distribution tool designed around a well architected framework is the right solution for both small and large networks. Also, since such a tool can provide most of the status processing required, the number of software packages to be distributed and frequency of distributions is not likely to diminish its usefulness.

An electronic distribution scheme may be designed around one of the following management strategies:

- Centralized Management
- Distributed Management

Each of these strategies have their inherent strengths and weaknesses. A centralized approach would provide the administrators with maximum control over all distributions. However, in case of large customer networks, it is likely to become extremely cumbersome to control and demand large amount of networking resources. Also, it will introduce a single point of failure when the central management station fails. A distributed management strategy however, will give the administrator maximum flexibility, consume a smaller amount of networking resources when configured properly and will have higher availability.

Best Strategy

In summary, the best strategy for software distribution is an electronic software distribution scheme designed around an architectural framework based on a distributed management strategy. The rest of this document will explore this strategy in a greater detail and propose an architectural framework to achieve the same.

Obstacles

The open systems strategy of the last decade or so has made possible heterogeneous networks supporting systems based on multiple operating systems including DOS, OS/2 and the UNIX system. Each of these operating systems has a number of versions with differing capabilities and feature sets. This wide variety of environments poses many obstacles for successfully implementing a comprehensive electronic software distribution mechanism. Any scheme that can not handle at least a majority of these environments is going to prove to be a serious problem for most administrators.

Given below are some of the more serious obstacles one has to overcome to implement a comprehensive software distribution solution:

1. Each of the various UNIX system dialects available on the market support disparate software packaging schemes. Therefore, the electronic software distribution scheme has to be able to either support these disparate schemes or provide mechanisms to enhance it to do so.
2. Both DOS and OS/2 platforms have no prescribed packaging schemes. When a workstation is being used as a personal computer, this issue does not cause much concern. However, when a workstation is being used for mission critical applications and one wants to remotely update its software configuration, a well defined packaging scheme is a necessity.
3. In all the environments, there are a number of software packages that assume that software will always be installed from a removable medium (floppy, tape, etc.). When performing installation from a remote site, all such packages will need to be re-packaged so that they may be installed from a local hard-disk, remote file system, etc.
4. Most software vendors do not yet support site/corporate license structures.
5. Since the concept of license servers is only now catching on, in most cases there is no guaranteed way of ensuring license enforcement.
6. Software distribution of large packages needs either high bandwidth networks or long time intervals. Some customer networks in existence today have not been designed to handle this additional traffic without significantly affecting network performance.
7. Due to the de-centralization of computing resources, each department in a corporation may have its own policies of how software should be installed.
   - Some departments may prefer to make sure that all the systems on the network have identical software configuration.
   - Some departments may prefer to allow each system to be configured separately.
   - Some departments may prefer to use a combination of the approaches mentioned above.

The issues listed above are only some of the major obstacles one needs to overcome. The next section in this document proposes a strategy that can not only handle these issues but also any new issues that may arise by providing mechanisms to enhance/customize the scheme to meet a specific environment.

Strategy

In the previous two sections, we discussed the various options available to perform software distribution and the obstacles we need to overcome to implement an electronic distribution scheme. This section discusses a strategy to implement an

electronic distribution scheme. It presents the strategy as a set of minimum requirements the scheme has to satisfy to be a useful solution. Given below are these requirements:

1. It needs to be a distributed solution so that a single network failure will not prevent software distribution.
2. It should be able to distribute at least UNIX system, DOS and OS/2 software.
3. To compensate for the lack of packaging schemes for DOS and OS/2 software, it should describe a packaging scheme for these packages and provide the required tools to convert DOS and OS/2 software into the prescribed packaging scheme.
4. It should provide scripting tools to support installation of interactive software packages using answer files created before hand.
5. It should provide mechanisms to customize the installation process of a software package so that each system may be customized after the package is installed. This should include support for various pre-installation and post-installation scripts.
6. It should support installation from removable media in the event of total loss of connectivity to any system on the network. It should also be able to capture changes in a system's software configuration due to such manual installations
7. It should support a hierarchical distribution structure so that the increase in network traffic due to software distribution can be kept to a minimum.
8. It should support at least the following types of software installation scenarios:
   - Installations initiated from a management site
   - Installations initiated at the system on which the software is to be installed
9. It should provide license metering features so that administrators can keep track of the number of copies of a software package currently installed on the network.
10. It should be able to provide a centralized view of software configurations of every system on the network.
11. It should provide reporting features to check for the presence or absence of a software package on a given system on the network.
12. It should provide scheduling features so that software distributions may be scheduled to take place during off-peak hours.
13. It should provide multi-level commit features for software distributions. Administrators should be able to first make sure that a software package is successfully delivered to one or more systems on the network before initiating installation.
14. It should have a friendly user interface so that inexperienced operators can monitor and activate distributions created by expert administrators.
15. It should allow the administrator to operate on groups of packages and groups of systems. For instance, an administrator should be able to request distribution of all CAD packages to all the systems in the engineering department by creating one single distribution request.
16. It should be able to install operating system upgrades.
17. It should be able to allow upgrades to itself.
18. It should be able to perform distributions over multiple network backbones and protocols.
19. It should provide mechanisms and APIs so that it may be customized to meet the specific needs of each networked environment.

Distribution Model

The distribution strategy discussed in the previous section may be implemented by using a distribution model that logically divides a customer's network into the following domains:

Source Domain Systems in this domain will have tools to prepare software packages for distribution.

Management Domain Systems in this domain will have the tools required create, control and monitor software distributions.

Destination Domain Systems in this domain accept software distributions from systems in the management domain.

For this model to be fully practical, each of the domains mentioned above have to be logical. That is, a single system may belong to one or more of the domains mentioned above.

Distribution Process

A distribution scheme architected around the strategy and model discussed above must implement the following distribution process in order to make full use of the strategy and the model:

- Software is prepared on one or more systems in the source domain.
- Once the prepared software is ready for release, it gets transferred to one or more systems in the management domain. This transfer can be initiated either from the source domain or from the management domain.
- Software received from the source domain is stored on one or more systems in the management domain for further distribution to the destination domain. Depending on the size of a network, there can be one or more systems in this domain.
- Software in the management domain gets transferred to one or more systems in the destination domain. This transfer can be initiated

either from the destination domain or from the management domain.

- Software received at the destination domain is installed and/or uninstalled. This process can be initiated either from the management domain or from the destination domain.

Architecture

To address all the issues discussed so far one would need to design an electronic distribution solution around a distributed framework that can also incorporate a hierarchical distribution strategy.

The architectural framework being proposed here is capable of accomplishing just that. Figure 1 provides a pictorial diagram of this architecture. As mentioned in the previous sections, this framework assumes that every network can be logically divided into source, management and destination domains.

Source Domain

The source domain contains all the components required to package, store and release software. To support each of these tasks a single component is defined by this architecture. Packaging of software is supported by the component referred to as a Source Packager. Any software prepared by the Source Packager needs to be stored for further processing. The Source Librarian is the component that provides the required storage features. Any of the stored software may be released for distribution by dispatching it into the management domain. The Source Dispatcher supports this functionality.

The components in the source domain mentioned above may be combined in a variety of ways. For instance in the case of a UNIX system, all the components may reside on the same system. Such a configuration is referred to as a Source Agent in this architecture. In the case of a workgroup environment

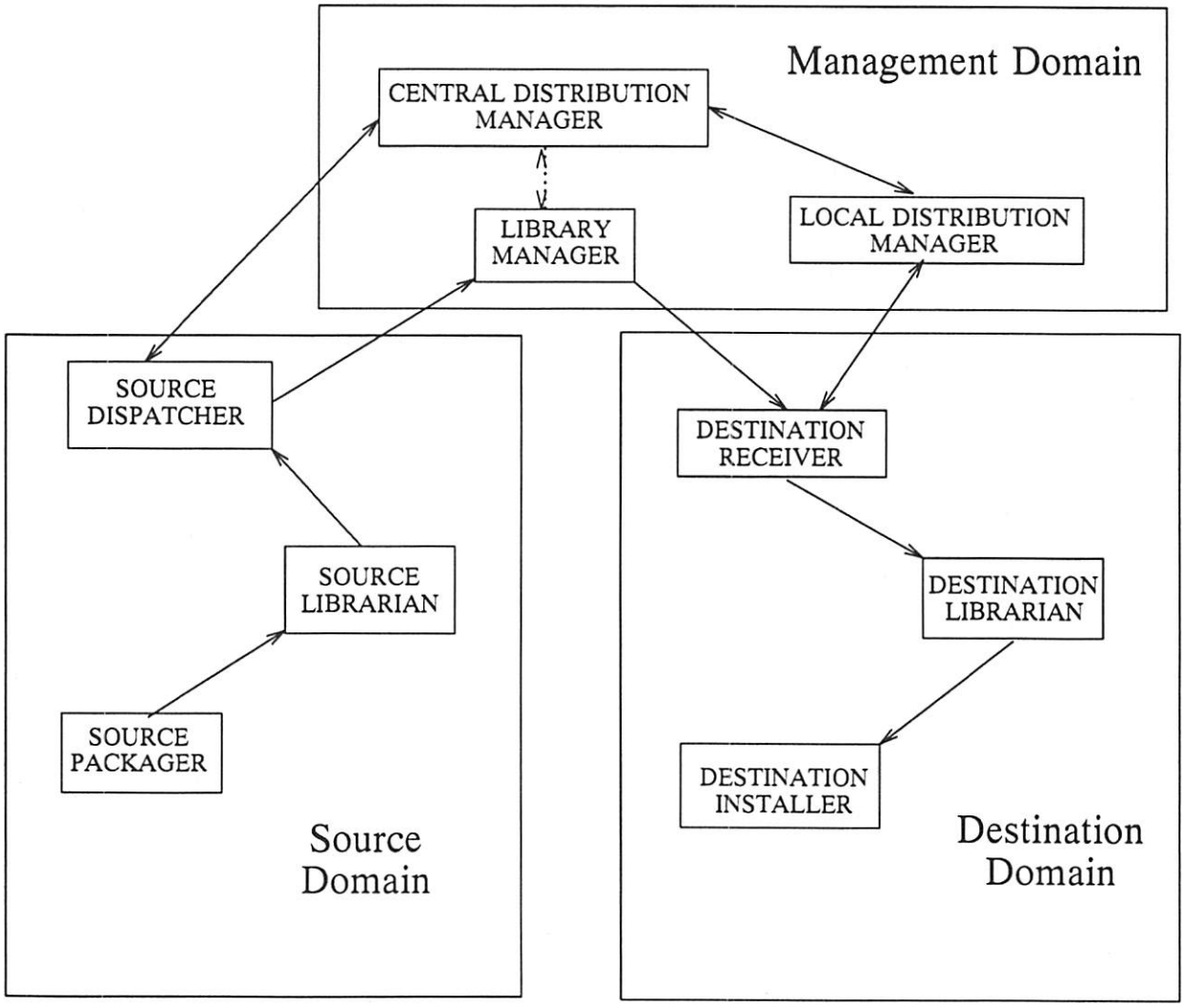


Figure 1: Architecture

containing DOS or OS/2 clients and a UNIX server, one or more clients may contain a Source Packager and the server may contain the Source Librarian and the Source Dispatcher. In such a configuration, the clients are referred to as Source Clients and the server is referred to as a Source Server.

Source Packager

A source packager provides the features required to

- Prepare software packages
- Mark prepared software packages for release
- Notify the associated Source Dispatcher about the released software.

Source Dispatcher

A Source Dispatcher provides the features required to

- Notify one or systems in the management domain about released software.
- Transfer released software to one or more systems in the management domain.

Source Librarian

A Source Librarian provides the features required to

- Maintain a software library that may be accessed by Source Packagers and Source Dispatchers.
- Accept and store software packages from a Source Packager.
- Allow a Source Dispatcher to access and retrieve software from the library.

Management Domain

Any software that is prepared in the source domain must eventually be transferred into the management domain so that it may be distributed to the rest of the systems in the network. Therefore, the management domain has to be able to store and distribute software. Also, this domain has to be able to receive software distribution requests from the systems in the destination domain and software release notifications from the systems in the source domain.

Therefore, the management domain needs to have a component that can support storage of released software. The Library Manager component of the management domain in this architecture supports this storage feature. You may recall from our previous discussions that an electronic software distribution strategy must make sure that it does not create too much network traffic. To be able to support this fundamental requirement, the architecture allows for the presence of one or more Library Managers on the network. The software released from the source domain may be transferred to some or all of these Library Manager depending on which systems in the destination domain need to be able to install the software.

Once the software is stored at the various Library Managers, there has to be a component that can process requests to distribute this software. The Central Distribution Manager component of this architecture provides the features required to do just that. The Central Distribution Manager can receive distribution requests from the destination domain and in response direct one of the Library Managers to transfer the required software to the requesting system. To ensure high availability of the distribution service, this architecture allows for the presence of one or more Central Distribution Managers.

Even though multiple Central Distribution Managers are allowed to be present on a network, it is still possible that some of the systems in the destination domain may not have the proper networking capabilities to reach these managers. To ensure that such systems can also receive electronic distributions, this architecture allows for the presence of intermediate systems between a requesting system and the Central Distribution Manager it needs to contact. This functionality is supported by what is known as Local Distribution Manager.

Each Local Distribution Manager can receive distribution requests from one or more systems in the destination domain and direct the Library Managers known to it to actually dispatch the software to the requesting system. Once again, to minimize the network traffic, there can be one or many Local Distribution Managers on a given network. When a Local Distribution Manager can not service a request on its own, it forwards it to the Central Distribution Manager associated with it.

Central Distribution Manager

A Central Distribution Manager provides the features required to

- Accept software release notifications from Source Dispatchers
- Accept software distribution requests from the local administrator or destination domain.
- Maintain software inventory of all the components in the architecture.
- Dispatch distribution requests to one or more Distribution Gateways.

Local Distribution Manager

A Local Distribution Manager provides the features required to

- Receive notifications, requests and commands from all other components.
- Forward notifications, requests and commands to other components.

Library Manager

A Library Manager provides the features required to

- Store released software received from Source Dispatchers and other Library Managers.
- Service distribution requests received from Distribution Manager by transferring the software to the destination domain.

Destination Domain

The collection of all the system in the network that may want to receive software distributions are referred to as the destination domain. The destination domain supports components to request, receive, store and install software. The component that requests and installs the software is referred to as a Destination Installer. The component that forwards the software requests from the Destination Installer into the Management Domain is referred to as a Destination Receiver. Since any received software will need to be stored before it can be installed, the architecture also supports a Destination Librarian.

The components in the destination domain mentioned above may be combined in a variety of ways. For instance in the case of a UNIX system, all the components may reside on the same system. Such a configuration is referred to as a Destination Agent in this architecture. In the case of a work-group environment containing DOS or OS/2 clients and a UNIX server, one or more clients may contain a Destination Installer and the server may contain the Destination Librarian and the Destination Receiver. In such a configuration, the clients are referred to as Destination Clients and the server is referred to as a Destination Server.

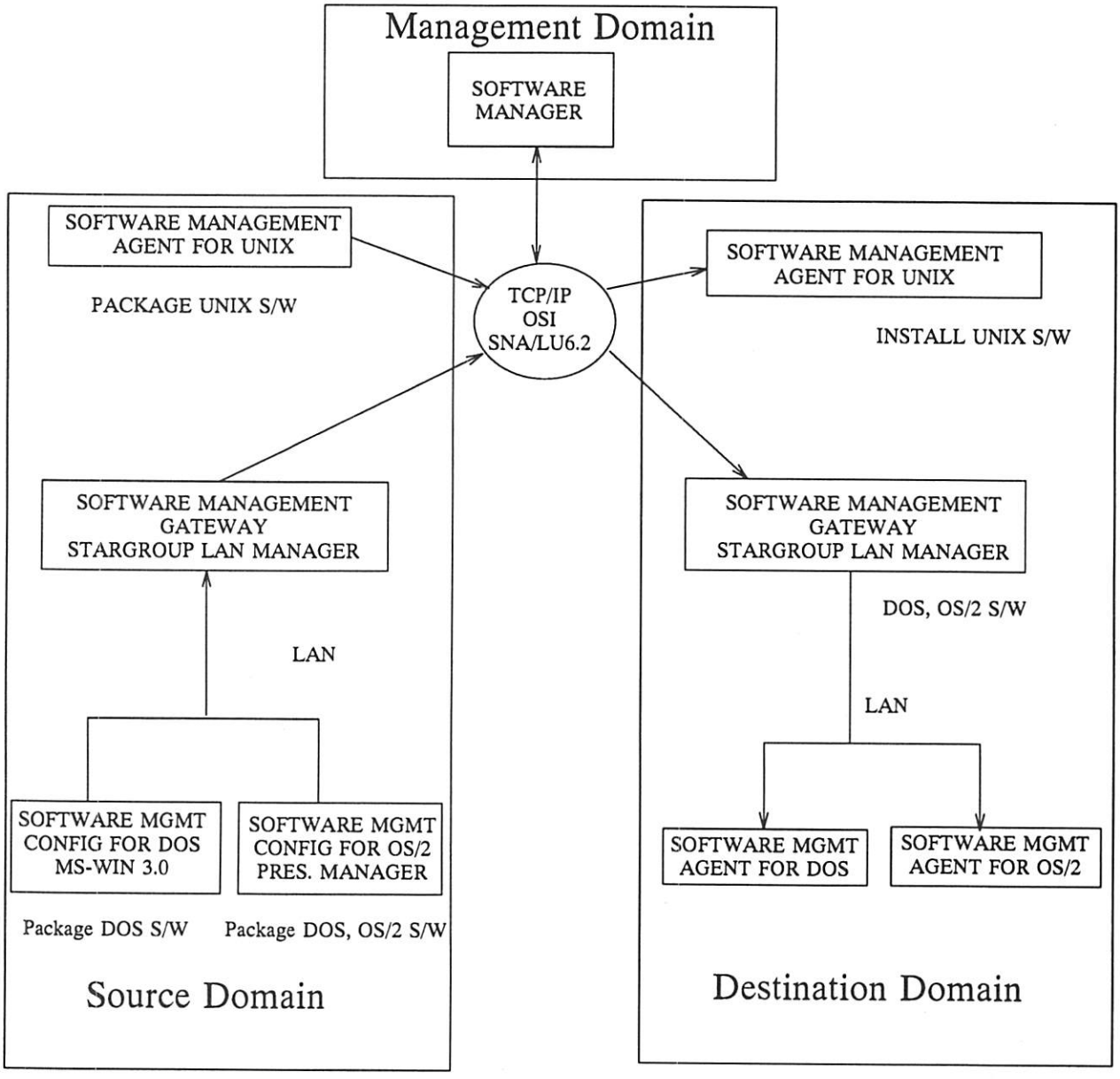


Figure 2: StarSENTRY

Given below is a short listing of the high level features each of the components needs to support.

Destination Receiver

A Destination Receiver provides the features required to

- Accept distribution commands.
- In response to distribution commands, accept software from Library Managers.
- Notify one or more Central Distribution Managers about received software.

Destination Librarian

A Destination Librarian features

- Maintain a software library for the destination.
- Allow a Destination Receiver and all its Destination Installers to access the library.

Destination Installer

A Destination Installer provides the features required to

- Service installation commands including:
  - Install
  - DeInstall
  - Remove
- Send notifications regarding changes in the software inventory.
- Accept installation requests from local administrator.

Implementation

The architecture detailed in the previous sections is the framework based on which the NCR corporation has developed the StarSENTRY Software Management Solution.

StarSENTRY is the name of the family of network and systems management products from the NCR Corporation. The StarSENTRY product suite includes as its base the StarSENTRY Systems Manager, a powerful standards based network and systems management platform. On this platform, users can implement a multitude of customer specific and generic solutions to handle their network and systems management requirements.

Some of the key applications in the StarSENTRY suite include:

- StarSENTRY Computer Manager – a UNIX systems management and administration application
- StarSENTRY Client Manager – a client management application that can manage work-group environments
- StarSENTRY Software Management Solution – an enterprise wide electronic software distribution solution

In addition, the StarSENTRY product suite also include various other network and systems management products which are beyond the scope of this document.

StarSENTRY Software Management Solution (SMS) is a reference implementation of the architectural framework discussed in the earlier sections.

StarSENTRY Software Management Solution (SMS)

SMS is a software distribution and management solution for the distributed computing environment. Figure 2 shows a view of the architecture of this product. SMS enables collection, storage, distribution and management of software packages to be performed from a central site.

The distribution aspect of the application involves the retrieval and distribution of software packages from and to computers at local and remote Local Area Networks (LANs), and over Wide Area Networks (WANs). The management aspect of the application involves version synchronization, geographic synchronization, and software inventory management. The collection aspect of the application involves retrieval of data from end nodes in the enterprise for central processing or sharing. There is also a central repository or library at the management station for cataloging and storing software packages that are installed in the distribution domain.

SMS is built around UNIX SVR4 management stations that communicate with remote UNIX, DOS and OS/2 systems over a variety of communication protocols and network topologies. The distribution of software is supported over LANs and WANs and is protocol independent. The current version of SMS supports TCP/IP, LU6.2 and OSI protocols over Ethernet, Token Ring, SDLC and X.25.

SMS is an integral part of the StarSENTRY suite of products. Therefore, it can also propagate any alarm conditions arising during software distribution to the StarSENTRY Systems Manager which is a standards based network and systems management platform.

The various products that are shown in the Figure 2 map into the architectural framework discussed earlier as follows:

Software Manager This product supports the functionality of a Central Distribution Manager and a Library Manager.

Software Management Gateway This product supports the functionality of a Source Dispatcher, Source Librarian, Destination Librarian and a Destination Receiver.

Software Management Configurator This product is only available on DOS and OS/2 clients and supports the functionality of a Source Packager.

Software Management Agent for UNIX This product supports the functionality of a Source Dispatcher, Source Librarian, and a Source Packager on a UNIX system in the source domain. It also supports Destination Receiver,

Destination Librarian, and Destination Installer in the destination domain.

Software Management Agent for DOS This product supports the functionality of a Destination Installer on DOS PCs. The SWMA for DOS works in conjunction with SWMG for UNIX.

Software Management Agent for OS/2 This product supports the functionality of a Destination Installer on OS/2 PCs. The SWMA for DOS works in conjunction with SWMG for UNIX.

Future

The first release of StarSENTRY Software Management Solution has implemented most of the components in the architectural framework detailed in this document. However, to validate the basic concepts, some of the flexibility allowed by the architecture has not been exploited. For instance, in the current release, there can only be a single Library Manager and it must be co-resident with the Central Distribution Manager. The next few releases of SMS are going to exploit the full extent of the architecture and also increase the variety of hardware and operating system platforms to which distribution can be achieved.

In addition, support for other file transfer schemes and transport protocols is also expected as early as in release 2.0.

The various Application Programming Interfaces (APIs) available in SMS will also be published so that one may create custom solutions around the SMS framework.

Given the flexible architecture and design of SMS, the authors are confident that support for newer operating system platforms will not need changes to the framework. Once the API's are published, support for other operating systems and communication protocols can be introduced by other vendors as well.

Summary

This document discussed the need for electronic software distribution in a networked environment and proposed an architectural framework and strategy to address the same. It also discussed a reference implementation of the suggested framework to provide a proof of concept.

There is a long way to go before all the software distribution needs of the distributed computing environment are fully met. However, with a well defined framework it is definitely an accomplishable task.

Availability

More detailed technical information about StarSENTRY Software Manager and any other products of the StarSENTRY suite may be obtained by contacting the primary author, Ram Vangala, by phone at 908-576-3710 or by e-mail at rvangala@sodium.att.com.

Author Information

Ram R. Vangala is the software architect and project leader for the StarSENTRY Software Manager in the Network and Systems Management business unit of NCR. He has been with AT&T since 1988. Prior to coming to AT&T, Ram worked as a consultant with various corporations around the world. He holds a Master of Science degree in Computer Science. Reach him via USMail at Ram R. Vangala; Consulting Analyst; Network Products Division; NCR; Room 1L-214; 307 Middletown Lincroft Road; Lincroft NJ 07738. Reach him via e-mail at rvangala@sodium.att.com.

Michael J. Cripps is one of the primary designers and developers of the StarSENTRY Software Manager. Prior to that he was a co-developer of the StarSENTRY Computer Manager. Prior to joining NCR, Michael had assignments with AT&T Computer Systems as well as other local small businesses. His specialties are in systems programming, graphical programming, and distributed computing. Reach him via USMail at Michael J. Cripps; Senior Principal Programmer Analyst; Network Products Division; NCR; Room 1J-210; 307 Middletown Lincroft Road; Lincroft NJ 07738

Raj is the R&D Director in the Video Telephone business unit of AT&T Consumer Products at Holmdel, NJ. Prior to that Raj was the Manager of Software Development in the Network and Systems Management business unit of Network Products Division in Lincroft, NJ. Raj was previously supervisor of product development in AT&T Bell Laboratories, where he was responsible for the development of several personal computer products. Raj holds Master of Science degrees in Computer Science and Mechanical Engineering and has been with AT&T since 1979. Reach him via USMail at Raj G. Varadarajan; R&D Director; Video Telephone SBU; AT&T Consumer Products; 4K-638 Crawfords Corner Road; Holmdel NJ 07733. Call him at (908)-834-1493 or use e-mail to contact him at raj@hocpa.att.com.

"Nightly": How to Handle Multiple Scripts on Multiple Machines with One Configuration File

Jeff Okamoto – Hewlett-Packard

ABSTRACT

A Unix system has a large number of subsystems that each have their own log files. In order to manage them, separate scripts are written and called from the *cron*(1m,8) scheduler. Since every system may not have every subsystem, every system's crontab file will be unique.

Once set up, the next problem is dealing with the output from the various cron jobs. As more machines are managed, more mail is received. It is then easy to inadvertently miss or accidentally delete a message that may contain an error message.

Nightly is a perl program that has two features that solve both of these problems. Its configuration file can specify on which hosts a maintenance script should or should not run on, and it can collect and return all the output from the various scripts and mail all of the output to a specified user or users. This means that only a single configuration file is needed for each system, so that a remote distribution mechanism like *rdist*(8) can keep all the systems synchronized. Additionally, only one piece of mail is delivered for each system.

Background

The group I belong to maintains over 400 Unix systems. Fortunately, most of these machines are diskless workstations, so the total number of systems is around 50.

Every night, every machine executes four to six shell scripts or programs from its root crontab entry. These scripts explicitly deal with various subsystems of each machine and would mostly consist of dealing with each subsystem's log file. Each one completes its task and sends a piece of mail to root. This amounts to a minimum of 200 messages per day, or 1400 per week.

In addition, each machine may run a different series of scripts, depending on the software that is on each system. Maintenance of each system's crontab entry becomes a chore, because each crontab may be unique.

Instead of executing jobs out of crontab, it would be better to execute a single program that could determine what needed to be executed on this system by reading a configuration file. This configuration file would then be identical on all systems and could be distributed to each system by *rdist*(8) or other such mechanism.[1] If this controlling program collected the output from its child programs and checked their return codes, it could, for instance, notice that all its child programs exited with zero status and send a single message that there were no problems. If one or more child programs exited with non-zero status or with zero status but produced some output, it could return all the output from those programs in one mail message.

To summarize, the "nightly" project (so named because it would run every night) had the following goals:

- Have one configuration file control what runs on every system.
- Have only one piece of mail generated per machine, whether it indicates positive or negative status.

Design

In order to accomplish these goals, certain functions would be required of *nightly*. Some of these were simple extensions of techniques that I already used and were easily converted. Others were newly created (and that can now be reused by other programs). These included:

- Interpreting an expression that determines what runs on which systems.
- Executing an arbitrary program, capturing its output, and receiving its exit status.

The first is necessary to interpret the configuration file's specifications. The second is the main portion of *nightly*'s functionality.

Other design features incorporated into *nightly* include executing the various scripts as any UID, and specifying to whom the output of any errors

[1] At Hewlett-Packard, we have an internal tool which uses a pull paradigm, as opposed to *rdist* 's push paradigm. It therefore does not need a potentially dangerous root .rhosts entry.

should be sent. These features are controlled in the configuration file through cpp-style directives.

Implementation

The first function involves evaluating a perl expression that has some minor differences. A hyphen that precedes the expression indicates that the script should not be run on the host or hosts that match the expression. Hostname matching is done left to right, with the first match taking priority, whether positive or negative. The rules expression must be carefully written so the desired effect is accomplished.

There is still a lot of work to be done on this function. It works and the code is readable, but is neither "elegant" nor general-purpose.

The second function, although easy to handle in C, was quite a bit more difficult to code properly in perl. The actual algorithm is quite standard, setting up an unnamed pipe, forking a child, dup'ing stdin, stdout, and stderr as necessary, then having the child exec the desired program while the parent waits in a select loop until it receives SIGC(H)LD.

Difficulties arose in correctly handling the death of the child. The child may die and a SIGCLD sent to the parent before all of the data from the child is read. However, inserting a conditional read from the file handle that holds the standard out and error of the child did not solve the problem; the condition was always true and the read ended up blocking. Therefore the code was removed, and the possibility of losing output from the child was accepted.

Another implementation hurdle was writing (or re-writing) the scripts that handle the subsystem log files. Daemons that do not use the *syslog*(3) mechanism may handle their log files in wildly different ways: some may open and close them when they need to write to them, others keep the file open all the time.

In order to safely copy the log files, it was necessary to examine the code for each subsystem and determine how each manages its log file. *Syslogd*(1m,8) turned out to be the most difficult to deal with. The algorithm adopted does the following: renames the syslog file(s), sends a SIGHUP to syslogd, then moves the log files to their storage location.

Cron is also an interesting subsystem. Cron opens its log file by calling *fopen*(3) in "a" mode and is then rebinds the stream to stdout. The method used for syslogd will not work because cron ignores SIGHUP. I was forced to accept the possibility of losing log file output between the time of copying the log file and truncating it to zero length.

The last obstacle to overcome was dealing with diskless clusters. On HP clusters, each diskless node (cnode) runs its own copy of the system daemons. However, there is one physical file system, and special directories contain what are called Context Dependent Files (CDF's)[2] that provide each system with its own unique view of the file system.

In order to lessen administrative maintenance, *nightly* is executed only on the cluster server. (In fact, *nightly* will exit if run on a cluster client). Moving the log files is easy, since any file, even a CDF, can be accessed from any cnode. However, to send signals to a daemon running on a cnode, it is necessary to remsh (or rsh) to each cnode.

Results

Nightly has been in use for some months on our standalone 9000/800 machines. It is also working on our standalone 9000/300 and 9000/700 machines. Nightly is undergoing final testing on our diskless clusters, both series 9000/300 and 9000/700. It has greatly reduced the amount of mail I receive.

In the course of creating *nightly*, I learned a great deal about how each of the various subsystems manages its log files. It was challenging to discover how to save each log file without losing any data in the process.

Improvements

In the newsgroup comp.lang.perl, Tom Christiansen posted a "one-line" script that verbatim could almost replace 20 lines of code. It just goes to prove Larry's motto.

A more general-purpose algorithm to specify and match host names would be preferable.

Biography

Jeff Okamoto graduated from the University of California at Berkeley with a BA in Computer Science in 1986. Since then, he has worked for Hewlett-Packard in what is now called the Palo Alto Information Technology Center as a systems programmer. In reality, he writes system administration tools in Perl, tries to keep up with USENET news, and occasionally even responds to users' requests. He can be reached at okamoto@ranma.corp.hp.com or at Hewlett-Packard; 3000 Hanover Street, Mail Stop 20CG; Palo Alto, CA 94304.

Appendix: Sample Configuration File

```
; This is a comment.
;
; Who will receive all the output?
;
#MAIL    root
;
; And now the actual scripts
```

[2]Despite its name, a CDF may be a file (/etc/inittab) or a whole directory (/usr/spool/cron/crontabs).

```
;
* /usr/local/etc/n.su
* /usr/local/etc/n.syslog
;
; Some fun stuff
;
; Runs on all machines except for not_me
-not_me,* /usr/local/etc/n.most_machines
12,-11-15,* /usr/local/etc/n.all_but_11,13,14,15
;
; Some other scripts
;
#UID adm
#MAIL adm
* /usr/local/etc/n.acct
```

One-Liner Replacement

In article 1992Aug3.203635.11211@news.eng.-convex.com, Tom Christiansen posted this "one-liner" perl script that takes arguments of the form "m-n" and prints the values between m and n, inclusive. The script is broken into its statements to improve readability.

```
($_='('.join(",",@ARGV).')')=~s/-/../g;
s/.*/"print join(' ',$&)"/ee;
```

Overhauling Rdist for the '90s

Michael A. Cooper – University of Southern California

ABSTRACT

The *rdist* program was first released as part of 4.3BSD UNIX. It was one of the first programs to address the area of automated software distribution in a timely, consistent manner in a highly distributed environment. Since its first release it has gained very wide use on almost every major UNIX platform. Despite all of its initial advances, *rdist* has some deficiencies which have not been addressed in any public release of the software until now.

This paper describes the past, present, and future of *rdist*. The program's history, operation, comparisons to other similar packages, and a number of possible future improvements in performance and functionality are described. A detailed description of *rdist* version 6, a major new version of the software that will be included in 4.4BSD, is presented. Version 6 *rdist* addresses many of the deficiencies of the original rdist, including significantly improved performance when updating large numbers of hosts, better error handling and avoidance, improved security, and cleanup of the actual source code.

Introduction

This paper describes major work done to a new version of *rdist* (version 6.0) that will appear in 4.4BSD. A look at the performance improvements in version 6, as well as possible future improvements, is discussed. The basic history and operation of *rdist* is described along with comparisons to other remote distribution packages.

The USC Environment

In order to provide better insight into the motivations behind the work described in this paper, it is helpful to describe the computing environment at the University of Southern California (USC) where most of the work was performed.

USC is a private University composed of two major campuses – University Park Campus (UPC) and Health Sciences Campus (HSC) – which are located some six miles apart.

The computing environment consists of about 3,000 hosts. This breaks down to about 1,000 UNIX machines (about 800 are Suns with the balance including NeXT, IBM RS6000, HP, and Silicon Graphics), 1,000 MS-DOS machines, 850 MacIntosh machines, and about 200 "other" types of machines. There are approximately 3,500 total "nodes" on USCnet, the campus LAN. USCnet physically consists of 12 cisco routers, 3 NSC routers, 1 Optical Data Systems (ODS) FDDI concentrator and myriads of various Ethernet and Fiber bridges and repeaters. Most of the routers are connected to an FDDI backbone. Twelve Sun SPARCsystems and one IBM RS6000 are connected to the ODS FDDI concentrator.

University Computing Services (UCS) is a service organization at USC which is charged with providing computing resources and support to the USC community. UCS centrally manages both centralized and distributed computing facilities. Approximately 800 UNIX hosts at USC are supported by UCS. Additionally, USCnet is also "owned", operated, and maintained by UCS. While UCS supports machines that are physically located in a centralized machine room, most computing resources are widely dispersed across the UPC and HSC campuses.

The current UCS support model for UNIX hosts is that all distributed and centralized resources are managed directly by a small group of people operating from a centralized physical location. Almost all major system administration work is done by UCS instead of by local system administrators. Each departmental machine under support has a designated Technical Contact. This person is usually a graduate student or professor who is responsible for performing "day-to-day" local system administration, such as resetting printer queues, setting up user accounts, and maintaining user disk quotas. UCS is usually the only party that has "root" (super-user) on machines connected to USCnet. The Technical Contact is provided with a special account which has access to a **set-uid** to "root" program which can run certain other programs as any valid user or group.

When problems or requests arise that are outside of the "day-to-day" privileges allowed the Technical Contact, they contact UCS via an electronic mail address called *action* or by calling the UCS Customer Service number.

History

History of rdist

Rdist was originally written in 1983-1984 by Ralph Campbell, then a graduate student at the University of California Berkeley. The only major release of the software was the version included in the 4.3BSD UNIX release in 1986. Since then, several major and minor security problems were

fixed by the CSRG group at Berkeley. Various sites and individuals in the research community have also modified *rdist* to increase functionality and performance. Those modifications, plus a number of bugs and suggestions submitted to CSRG, have been evaluated and addressed in *rdist* 6.0.

History of rdist at USC

The first experiences with UNIX at USC where with a couple of DEC VAX-11/750's running 4.2BSD UNIX back in 1984-1985. It was apparent from the start that some type of software was needed to maintain distributed computing resources. However, nothing was available at that time that really addressed the issues involved with maintaining thousands and thousands of files on a large number of machines (two dozen UNIX hosts was considered a "large number" back then). When we purchased our first workstations (Sun-3/50's) in early 1986, there was still nothing to maintain all the distributed files. We were still muddling along with simple shell scripts for our thirty UNIX hosts. By the time *rdist* was released with 4.3BSD, and shortly afterwards with SunOS, we were really starting to struggle to maintain our 100 UNIX hosts.

The release and subsequent Great Discovery of *rdist* in 1986-1987 came as the number of UNIX workstations started doubling every year. Over the year or so following the Great Discovery, the number of supported machines continued to dramatically increase. *Rdist* all too quickly started showing signs of significant deficiencies in performance. The goals of the part time work that started in 1988, were to improve efficiency, reliability, and performance, in order to support the ever growing number and variety of UNIX machines. Looking back at all the work done, it would have been far better in the long run to have completely re-written *rdist* from scratch.[1]

The first major change made was to add support for updating multiple hosts in parallel. Despite a large number of other changes since then, this one item still remains as the single largest performance increase realized to date.

Release History

The first version of *rdist* to be released was the version included in 4.3BSD. This version spoke version 3 of the *rdist* protocol.

The first version of *rdist* to be released externally from USC was version 5.0 (protocol version 5) in 1991. This version never got farther then *beta* testing at a small number of external sites. During the *beta* test, a copy of all the submissions regarding *rdist* bugs, suggestions, and improvements sent to the CSRG group at UC Berkeley, was received. This resulted in a significant number of changes that obsoleted version 5.0 and spawned version 6.0 (protocol version 6), which is the first version of *rdist* from USC to make it into general distribution.

Description of Rdist

What It Does

Rdist is a program that maintains identical copies of files over multiple hosts. It preserves the owner, group, mode, and modification time (*mtime*) of files, if possible, and can update programs that are executing. *Rdist* reads commands from a *distfile* to direct the updating of files and/or directories similar to how *make* reads recipes from a *Makefile*. The syntax of *distfile* is described in the *rdist*(1) man page in Appendix A.

What It's Good For

Rdist is a general purpose program which can be utilized for a number of purposes. It provides an excellent mechanism for maintaining consistent versions of files that contain text, binary, or other type of data. This is invaluable for maintaining the same version of operating system files on a multitude of hosts.

A number of sites use *rdist* in place of Sun's Network Information Service (NIS, aka YP) to maintain consistent, distributed copies of */etc/hosts*, */etc/passwd*, */etc/group*, */etc/services*, and others[1].

At USC, *rdist* is normally used to maintain consistent versions of operating system files as well as third party software packages such as X11, GNU Emacs, and many others. It is often used to perform minor operating system upgrades such as upgrading hosts from SunOS 4.1.1 to SunOS 4.1.2.

How It Works

Rdist parses a file called *distfile* which contains a description of what needs to be done. Once this file is parsed, *rdist* will open connections to multiple clients in parallel using the *rcmd*(3) interface[2]. If a connection attempt fails, or a running session fails, no further connection attempts are made during that instance of *rdist* in order to avoid slowing down updates to other hosts.

The *rcmd*() routine makes a connection to the *rshd*(8c) program on the remote host. The *rshd* program in turns checks to see if the user is authorized via the ".rhosts" mechanism. If the authorization succeeds, then the user's login shell is started, using the command:[3]

```
rdistd -S
```

The user's login shell searches for the program

[1] Hind sight is always 20/20.

[2] The *rcmd*(3) interface is the same one used by the *rsh*(1c) command.

[3] The -S option is required to avoid accidental execution by actual users since *rdistd* usually resides in the normal user's search path.

rdistd in the user's search path (**$PATH**) and executes it if found. *Rdistd* now sends a message back to *rdist*, through the *rsh* connection, requesting *rdist* to send its protocol version number. *Rdistd* reads the version number and decides whether it supports that protocol version. If it cannot support it, it will return an appropriate error message to *rdist* and exit. Otherwise *rdistd* starts accepting commands from *rdist*.

After negotiating the protocol version with *rdistd, rdist* now sends global configuration information to *rdistd*. This information includes the name of the host *rdist* is running on[4] as well as any global parameters such as minimum amount of free space and files that must exist before a file will be installed. Once this configuration information is successfully sent, *rdist* starts checking files as specified in the *distfile*. The actual protocol involved in this, is beyond the scope of this paper.

Methodology of Operation

The methodology behind the operation of *rdist* is one of a dictator forcing his/her view upon his/her subjects regardless of the subject's will. This works well in environments where there is centralized management which has almost absolute control over the distributed environment. A centralized machine (or machines) can utilize *rdist* to maintain a large number of distributed machines.

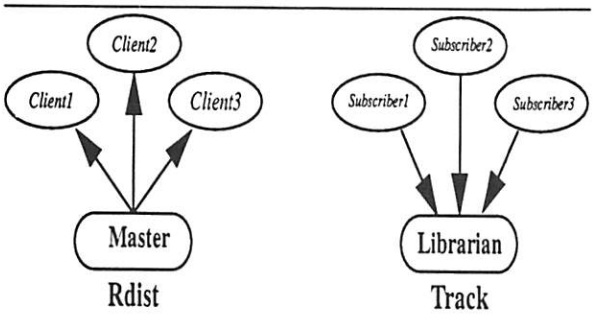


Figure 1

Another major benefit to this model is in security. Maintaining a tight security leash on a few centralized machines which *rdist* to some number of other machines can be an excellent mechanism for automatic detection and correction of security intrusions. The corruption of system programs is a typical method used by many intruders to gain further access and privileges. This type of attack is easily detected and corrected with the proper application of *rdist*. Obviously, the master *rdist* host(s) must be carefully guarded against being compromised, or else all your client machines may become compromised.

[4] *Rdistd* does not know the name of the host *rdist* is running on because *rshd* does not provide any mechanism for obtaining this information. Thus, the host name is for informational use only and should never be used for any kind of authentication.

Comparisons

Comparison to Track

The *track* system as described in [2] is another automatic software distribution package similar in capabilities to *rdist*. It differs significantly in its basic architecture and the methodology behind it.

Architecture Differences

The main architectural difference between *rdist* and *track* is in the *track* model of *pulling* files from a centralized *Librarian* host down to *Subscriber* hosts. The *Subscriber* host has a *subscription* file which contains lists of software *packages* and the *Library* host that they reside on. The *Subscriber* host is responsible for initiating contact with the appropriate *Library* host(s) in order to update and/or verify *packages*.

This model works well in environments where distributed hosts are administered by mostly autonomous local system administrators. In environments where centralized control and security are of primary importance, the *rdist* model of *pushing* files out to client machines is more in line with those priorities.

With *rdist*, the *master* distribution host is the only point where configuration information must be maintained. The *track* model is that each *Subscriber* must maintain the list of packages it wants to receive. The *Librarian* maintains the details of each package, which includes the package contents and the *Subscribers* authorized to receive the package. A misconfigured *Subscriber* host may go undetected for a long period of time. Automating the distribution of the subscription information from a central *Library* host may solve some of the problems. However, this would remove the local autonomy aspect of *track* and still not provide for the automatic detection and correction of security intrusions that is inherent in the architecture of *rdist*.

Transport

Track also differs from *rdist* in the transport layer used to communicate between client and server. *Rdist* uses *rsh* (via the *rcmd*() library routine) as its transport layer. *Rsh* in turn is implemented on top of TCP/IP. *Rdist* does **NO** authentication itself, instead relying on *rsh* to perform the authentication. *Track*, on the other hand, is implemented directly on top of the TCP/IP layer. The client and server communication through a "well-known", privileged TCP port. The server is responsible for doing all authentication.

One of the benefits of using *rsh* as the transport layer, is in portability to other platforms and network protocols. Since most every 4.X BSD UNIX based platform supports *rsh* there is no transport layer in *rdist* to port. If *rsh* is ported to utilize another network protocol, *rdist* can also automatically make use of it.

On the down side, using *rsh* as a transport layer severely limits the authentication performed for an application. Currently *rsh* relies on the often insecure method of IP host address and privileged ports as a means of authentication. This does not provide for a secure means of *rdistd* obtaining the actual identity of the user or host running *rdist* on the remote host.

The implication here is that *rdist* cannot provide a *subscription* service as *track* does. The reason for this lies in *rsh* which hides the actual identity of the caller. This means *Rdistd* doesn't know the authenticated identity of the caller, which is a requirement of most *subscription* oriented services in order to restrict access of certain files and commands to certain callers. *Track* can authenticate the caller's identity since it listens on its own TCP port and therefore "knows" the identity of its caller in the same manner that *rsh* does.

The eventual solution to the caller identification problem in *rdist* is probably *Kerberos*. Currently *rdist* does not use *Kerberos* since there are still a few vulnerabilities in the *Kerberos* protocol and because its use is not yet wide spread. It should be a trivial matter to add *Kerberos* support to *rdist* at a later time.

Changing *rdist* to eliminate the *rsh* layer and use its own well-known TCP port was considered at one point. The only major benefit gained would be the ability to implement a subscription service. The amount of work and additional complexity required for this was deemed excessive in the face of the eventual use of *Kerberos*. Those sites requiring the *pull file* model can use *track*. Those sites requiring the *push file* model can use *rdist* as currently implemented.

Feature Differences

There are some basic feature differences between *rdist* and *track*. One of the most notable is *track*'s use of a *state database* on the *Librarian* host, which is implemented as a "flat" ASCII text file. The database contains state information for files in the *Library*. This information is basically the file type, the path name of the file, and the *currentness* of the file. The *currentness* is defined based on the type of file. The *currentness* of regular files is the file's modification time. For directories its based on the owner and group of the directory. For symbolic links it is the pathname the link points to.

The state database is periodically generated by running a command either via an automatic scheduler such as *cron*, or manually from a users shell. The *track* state database generator provides for only coarse updates by checking all the files listed in the *Library* subscription list. This model works sufficiently if updates of files that are in the *Library* are infrequent. In a volatile environment such as USC, which has frequent (daily) updates to many different "sections" of a *Library*, a coarse grain state database generator involves significant overhead which greatly increases the time to update files. Experiments with using the same type of state database generator with *rdist* showed an overall *increase* in the time it took to verify a large number of *volatile* files.

The *rdist* master host at USC which updates the most files is a Sun SPARCsystem 490 with 128MB main memory and 15GB of IPI-2 disk. Using a coarse grain generator similar to *track*'s, it took approximately 8.3 hours to generate a database for some 512,000 files. Once generated, an *rdist* session reading from this database instead of doing a *stat*() call for every file for every client, took 3.3 hours to do an *rdist verify* of all files on 103 hosts, with 4 clients being checked simultaneously. The same setup *without* using the state database took 4.5 hours. The total aggregate time with the state database is 11.6 hours compared to 4.5 hours without. Notice that the *without* number is not significantly higher than the verify time of the *with* number. This is most likely due to the kernel caching the inode information which is advantageous to the simultaneous updates of clients since they are usually checking the same files at roughly the same time. A possible solution to this problem is to implement a fine grain state database generator which is discussed later in the paper in the **Future Directions** section.

Other Differences

Other features that are present in *rdist* but missing in *track* include *track's* inability to detect when the owner, group, file mode, and/or contents of a regular file has changed, but the modification time (*mtime*) has not. Only the *mtime* is checked to determine if the file is current or not.

Track also lacks a regular expression facility for the *exception* field in the *subscription* file. This could be corrected by using one of the many regular expression libraries available, such as the Berkeley regex (*re_comp()/re_exec()*) routines which are used by *rdist*.

Comparison to Ninstall

Ninstall as described in [3] is similar in architecture to *track*. It provides the same model for *pulling* files from other hosts and installing them on the machine it was run on. It differs from *rdist* along the same lines as *track*.

Comparison to Ru

Ru as described in [4] is also similar in concept to *track*. It differs in architecture with *rdist* in a manner similar to *track*. The fact that its implemented as a small set of shell scripts shows that an automated software distribution system need not be complex to provide minimal functionality.

Deficiencies of Original Rdist

The original *rdist* had a number of deficiencies that are described below. It should be noted that for all its flaws, it made for a very useful and pioneering tool which helped pave the way for *rdist* version 6 and several others.

Serial Host Updates

The original *rdist* would only update one single host at a time. Updating a large number of hosts was extremely slow and inefficient because of this. One very slow host, or a connection that hangs, would also delay updates for all other hosts, possibly indefinitely.

In 1983, when *rdist* was originally written, CPU speed and I/O bandwidth was still relatively slow and kernel *inode* caching techniques were somewhat primitive. This made applications such as *rdist* extremely disk bound in performance. The benefits of parallel *rdist* updates were negligible when faced with these limits. Since the original version was written, significant increases in CPU speed and I/O bandwidth, as well as advances in kernel *inode* caching techniques, have made parallelizing once disk-bound applications, such as *rdist*, very appealing.

Hangs

The original *rdist* never attempted to avoid hanging. A remote host could get into a state where it never completed an *rdist* session, thus hanging all other host updates.

Security Problems

The original *rdist* had one program, *rdist,* which was **set-uid** to "root". This meant that both the client and server *rdist* processes had to run as "root" which led to a number of security holes.

Bad Coding Style

The coding style used in the original *rdist* left even some of the best C programmers mystified as to many of the its inner workings. One of the most notable problems was in the use of hardwired protocol values. Instead of using *#define* definitions for the *rdist* protocol commands, the actual values were coded in multiple places (usually at least twice – once in the client modules and once in the server modules) throughout the source code. This made following, let alone extending, the protocol logic a feat not for the meek.

The repetition of redundant code often served to cause confusion when changes were made to one part of the code, but not another. The nonuse of "standard" C library routines also led the code being to illegible, redundant, and non-portable.

The combination of the client and server programs into one program made the logic flow cluttered and unnecessarily complex in many areas. This often resulted in vulnerabilities in security.

The rampant use of *goto*s and use of large routines with many levels instead of smaller, simpler functions, also made logic flow unnecessarily difficult to follow.

Miscellaneous

The mode and ownership of files was never checked for *currentness* in the original *rdist.* Only when a file's *mtime* was found to be out of date and a file was updated, was the file's mode and ownership set.

The original *rdist* could not handle rdisting a directory on the local machine to a symlink on a remote machine. For example, suppose you have a directory */foo/bar* on machine A and machine B. If you replace */foo/bar* on machine B with a symlink and then *rdist* from machine A to machine B, *rdist* would fail (with an error) to update */foo/bar* on machine B.

New Features of Rdist

Many new features have been added to *rdist* 6.0 since the release of the original in 4.3BSD. Some of the highlights are detailed below.

Parallel Host Updates

Multiple target hosts are now updated in parallel. This can dramatically improve the update time for a large number of hosts. The effective number of concurrent updates is limited by the CPU and I/O bandwidth of the host running *rdist.* The value of 4 concurrent updates has proven an adequate default value for many different platforms. See the **Performance** section for a discussion of this.

Improved Error Handling and Avoidance

Timeouts

There are a number of ways that *rdist* can hang if not properly avoided. One such case is when a host goes down and does not reboot. If this occurs while *rdist* is doing a *read*() waiting for response from the unreachable host, the TCP/IP kernel layer does not pass up a failure error unless the remote host comes back up on the network. A similar situation can occur if the remote host is a NIS client and loses contact with an NIS server.

Rdist avoids hanging by setting timeout alarms via *alarm*() and *signal*(). Before calling *read*() to read a response from the remote host, an *alarm*() is set for a given period of time (10 minutes by default). If no response has been received at the end of the timeout period, the session is marked as *failed* and the child *rdist* process for that particular host exits with an error status. The parent *rdist* process then proceeds to the next host to update.

Avoid Retries

If a connection to a host fails at any point for any reason, no further attempts are made to contact the host during that instance of *rdist.* This avoids having a host that is unreachable, or not responding

properly, from slowing down, or even preventing, the updates of other hosts.

Error Messages

Local and remote error messages are distinctly marked as such for better clarity as to the origin of the error. This makes determining the cause of the error much simpler.

Free Space Checking

The amount of free space and/or free files/*inodes* can optionally be checked to avoid filling up a filesystem. Before actually installing or updating a file, *Rdist* will calculate whether the update would exceed the minimum amount of free space and/or inodes as specified on the command line. If the minimum space would be exceeded by the update, no update is performed and an error message is displayed. This allows *rdist* to be less intrusive by preventing it from filling up a filesystem.

The cost for this avoidance is increased overhead resulting from additional system calls. The *statfs*() system call is called for most every update of a file. The calculation and call are performed after the *currentness* check but before the actual install/update is performed.

Split Client and Server

The client and server portions have been split into two distinct programs, *rdist* and *rdistd*, respectively. This lowers the risk of security vulnerabilities since the server *rdistd*, does not need to be **setuid** to "root". It also allows for greater ease in maintaining different protocol versions of *rdist*.

Major Code Cleanup

Most sections of the source code have been cleaned up, and in many cases, re-written. A majority of this cleanup was to make the code more understandable and to clarify the underlying *rdist* protocol itself. Many routines were converted to use standard system library routines. The code has been ported to a number of different UNIX platforms, including several System V.3 based systems, with relatively little changes.

Miscellaneous

All reported security holes have been fixed.

Modes of files and directories are now checked. If they are different, the entire file on the remote machine is updated. This is done instead of just changing the mode and/or ownership in case the file was compromised in some manner. This is usually quicker than doing a full binary comparison of the local and remote files.

Rdist now has the ability to rdist a directory to a symbolic link. The original rdist could not handle this.

Rdist will optionally check whether a file resides on a NFS and/or a read-only filesystem. If so, no update will be done unless explicitly specified for that host.

A general protocol command was added during the initial connection negotiation to support setting certain parameters. These parameters include the name of the host running *rdist*, the minimum amount of free space, and the minimum number of free files/*inodes* that must be available on a filesystem for a file to be updated. The host name is used for internal logging purposes by *rdistd* and is also used in setting the process arguments to show where the host is being rdist'ed from.

Support was added to do buffered reads. Instead of reading one character at a time from the remote host like the original *rdist* did, it now attempts to fill a full 8 kilobyte buffer whenever possible. This can lead to measurable reductions in system overhead due to the reduced number of kernel *read*() calls needed.

All configuration information is now stored in two places, **config.h** and **Makefile**. Porting to a new platform usually only requires slight modifications to these files.

Performance

Parallel Updates

The most significant performance improvement seen in version 6 of *rdist* is in parallel updates. This can be evaluated by running a series of tests varying the number of simultaneous updates. Table 1 shows the results of two such tests. The **Number** column indicates the number of simultaneous clients being updated, **Time** indicates elapsed "wall-clock" time in minutes, and **%CPU** indicates the percent of the CPU used.

Number	Test 1		Test 2	
	Time	%CPU	Time	%CPU
1	756	10	1600	18
2	309	24	1012	30
3	254	31	713	48
4	175	48	873	36
5	217	40	413	73
6	195	46	431	71
7	198	42	446	69
8	207	40	432	73
9	208	40	510	63
10	200	42	568	58

Table 1

The test used in both cases was to run *rdist* in *verify* mode, varying the number of simultaneous clients being updated between 1 and 10. Each test used a different *rdist* master and a different set and number of files. All machines involved with the test, including the master *rdist* hosts, where running their normal work load. This accounts for some discrepancies in the results.

Results of Test 1

The *distfile* used in **Test 1** is used at USC to maintain a body of some 10GB of software. The test was run on a Sun SPARCsystem 490 with 128MB of RAM and 15GB of IPI-2 disk. A total of 29 hosts were listed in the *distfile* as client hosts.

The results for this test indicate that the optimum number of simultaneous updates is 4. The elapsed time drops off most significantly between 1 and 2 clients, in comparison to any of the other numbers. This is likely due to the kernel caching the *inode* information that occurs when *rdist* calls *stat*() which is where *rdist* incurs most of its overhead. Since the two *rdist* processes are *stat*()'ing the same set of files at roughly the same time, the kernel only has to read the *inode* information once from disk in most cases.

The amount of CPU used in **Test 1** levels off around 42% starting around 4 parallel updates. This may also be an indication that the kernel is able to use cached *inode* information most of the time. This ability is probably due to the large amount of RAM (128MB) in the master machine used for this test, which allows more information to be kept in the kernel *inode* cache.

Results of Test 2

Test 2 uses a *distfile* that is used at USC to maintain the SunOS 4.1.2 operating system files which include */sbin*, */usr/5bin*, */usr/5include*, */usr/5lib*, */usr/bin*, */usr/etc*, */usr/kvm*, */usr/share*, and */usr/ucb*. The *rdist* master host used was a Sun SPARCstation ELC with 8MB of RAM and 300MB of SCSI disk. A total of 245 hosts were listed in the *distfile* as client hosts.

The results of this test show that 5 simultaneous updates is the most optimal value for elapsed time. However, the percent of CPU required for 5 updates is 73% as opposed to just 36% for 4 parallel updates. This indicates the presence of a threshold in the system of some type. Most likely the threshold is the system running out of physical memory which results in increased paging rates and keeping less *inode* information cached in the kernel.

Future Work

There are a number of possible avenues for future work to follow which may lead to increased performance and functionality.

State Database

The implementation of a state database, and a corresponding fine grain state database generator, is a likely source for further performance improvements. The cost of updating the state database must be lower in overhead than simply checking the state of each file during the actual update session as is done in the current implementation. Such a generator must have the ability to check and update specific files, directories, or packages of software. In this way, a system administrator who updates a file or entire package of software, could manually run the generator specifying the newly updated files or packages to check. Additionally, the most frequently changing packages could be automatically checked hourly or daily with a full check of everything on a weekly basis.

Another area of interest for the state database problem is the underlying database itself. Flat files are very inefficient to update and search if they contain large numbers (large being greater than 100,000) of entries (files) and are very hard to update quickly in a fine grain manner.

The *dbm*(3) routines also do not scale well to large numbers of entries. The new Berkeley hash database package [5] may be one possible solution. There are certainly a number of commercial databases that could handle the data, but the financial cost is, of course, prohibitive.

Message Handling

Message handling is one area which requires some attention. The quantity of output from *rdist* can be quite voluminous as well as useless in content. The amount of output that can result in just one new software package, such as MIT's X11R5, can be hundreds of thousands of lines. Human perusal of so much output is tedious, boring, and is usually ignored.

The destination of the messages could also be improved. Currently, all errors that occur are displayed in the normal output along with the normal messages. Major errors that *rdistd* encounters, are logged via the *syslog* facility.

The current *rdist* includes a filter written in *Perl* to put the output in a more human readable format. It does nothing to reduce the amount of data nor provide any filtering of the type of messages seen.

To address this problem, a comprehensive overhaul of the message handling system in *rdist* is needed. All messages need to be classified by type and assigned a severity level which should look something like those of *syslog*. The message handling system will also need to be changed to support a variety of logging destinations such as files, *syslog*, electronic mail, and perhaps a general purpose facility to allow piping to an arbitrary user specified command. A better filter program on top of all this could also increase the readability and amount of data shown.

File Checking

Another area for possible performance improvement is in checking actual file contents. The current *rdist* offers the option of doing a byte-by-byte comparison of files. This is implemented by having *rdist* send a full copy of a file over to *rdistd* and then

doing a byte-by-byte comparison. In the case of large files and relatively slow or congested network links, this can be prohibitive. Implementing a checksum test would address this issue. A checksum would be performed for the files on the client and server by both *rdist* and *rdistd* and the results compared. The file would then only need to be transferred if it's out of date and *rdist* is in update mode.

Polling the Rdist Server

An interesting feature that was added to the original 4.3BSD *rdist* by the Ballistic Research Laboratories (BRL) was the ability of a client *rdist* host to *poll* an *rdist* master host. This allows the client to be rdisted to on-demand as opposed to when the master wants to.

The way it works is the client runs *rdist* with the *poll* option specifying the name of the master *rdist* host to contact. The client *rdist* then uses the same *rcmd*() interface as the master does to open a *rsh* connection to the master. The command run via *rcmd*() tells the *rdist* server to go into *polling* mode and also specifies the name of the client doing the poll. The master host then *reverses* the connection, parses the *distfile* in the current directory (which should be the home directory of the user the server is running as), and proceeds with a normal *rdist* session for just the polling host.

The polling feature is not present in version 6.0 of *rdist*. The changes from BRL are not easily incorporated into version 6.0 due to the separation of the client and server processes into separate programs. It's also not clear if this is really that desirable. In order for it to work, the server program, *rdistd*, probably needs to be **set-uid** to "root", which removes one of the more desirable features of version 6 *rdist*.

Conclusions

The system described above has gone through a number of internal releases since work began 4 years ago. It currently is used to support some 800 UNIX machines at USC. While many new features have been added since the original 4.3BSD version, no further significant performance gains have been realized since the initial support for parallel updates was added. Adding support for a state database and fine grain database generator hold hope for another leap in performance which we hope will be able to sustain the current growth rate of UNIX machines here at USC and elsewhere.

Availability

Rdist is available via anonymous ftp from host **usc.edu** as **/pub/rdist/rdist.tar.Z**.

References

1. Bjorn Satdeva and Paul M. Moriarty, "Fdist: A Domain Based File Distribution System for Heterogeneous Environment," in *LISA V Conference Proceedings*, pp. 109-126, USENIX, San Diego, CA, September 30 – October 3, 1991.
2. Daniel Nachbar, "When Network File Systems Aren't Enough: Automatic Software Distribution Revisited," in *USENIX Conference Proceedings*, pp. 159-171, USENIX, Atlanta, GA, Summer 1986.
3. Mike Rodriquez, "Software Distribution in a Network Environment," in *Large Installation System Administrators Workshop Proceedings*, p. 20, USENIX, Philadelphia, PA, April 9-10, 1987.
4. Tim Sigmon, "Automatic Software Distribution," in *Large Installation System Administrators Workshop Proceedings*, p. 21, USENIX, Philadelphia, PA, April 9-10, 1987.
5. Margo Seltzer and Ozan Yigit, "A New Hashing Package for UNIX," in *USENIX Conference Proceedings*, pp. 173-184, USENIX, Dallas, TX, January 21-25, 1991.

Author Information

Michael Cooper has been working on UNIX systems for 10 years. He has worked in the Research, Development, and Systems Group of University Computing Services at the University of Southern California since 1985. Reach him via U.S. Mail at: University Computing Services; University of Southern California; Los Angeles, California, 90089-0251. Reach him electronically at mcooper@usc.edu.

NAME

rdist – remote file distribution client program

SYNOPSIS

rdist [**–bDFhinORrsvwxy**] [**–A** *num*] [**–a** *num*] [**–d** *var=value*] [**–f** *distfile*] [**–M** *maxproc*] [**–m** *host*] [**–t** *timeout*] [*name ...*]

rdist –bDFhinORrsvwxy -c *name ... [login@]host[:dest]*

rdist –Server

rdist –V

DESCRIPTION

Rdist is a program to maintain identical copies of files over multiple hosts. It preserves the owner, group, mode, and mtime of files if possible and can update programs that are executing. *Rdist* reads commands from *distfile* to direct the updating of files and/or directories. If *distfile* is '–', the standard input is used. If no **–f** option is present, the program looks first for 'distfile', then 'Distfile' to use as the input. If no names are specified on the command line, *rdist* will update all of the files and directories listed in *distfile*. Otherwise, the argument is taken to be the name of a file to be updated or the label of a command to execute. If label and file names conflict, it is assumed to be a label. These may be used together to update specific files using specific commands.

The **–c** option forces *rdist* to interpret the remaining arguments as a small *distfile*. The equivalent distfile is as follows.

```
( name ... ) -> [login@]host
        install   [dest] ;
```

The **–Server** option is recognized to provide partial backward compatible support for older versions of *rdist* which used this option to put *rdist* into server mode. If *rdist* is started with the **–Server** command line option, it will attempt to exec (run) the old version of *rdist.* This option will only work if *rdist* was compiled with the location of the old rdist (usually either */usr/ucb/oldrdist* or */usr/old/rdist)* and that program is available at run time.

Rdist uses the *rcmd(3)* interface to access each target host. *Rdist* will attempt to run the command

rdistd -S

on each target host. *Rdist* does not specify the absolute pathname to *rdistd* on the target host in order to avoid imposing any policy on where *rdistd* must be installed on target host. Therefor, the user on the target host that *rdistd* runs as, must have *rdistd* somewhere in there **$PATH.**

OPTIONS

–A *num* Set the minimum number of free files (inodes) on a filesystem that must exist for *rdist* to update or install a file.

–a *num* Set the minimum amount of free space (in bytes) on a filesystem that must exist for *rdist* to update or install a file.

–b Binary comparison. Perform a binary comparison and update files if they differ rather than comparing dates and sizes.

–D Enable copious debugging messages.

-d *var=value*
Define *var* to have *value*. This option is used to define or override variable definitions in the *distfile*. *Value* can be the empty string, one name, or a list of names surrounded by parentheses and separated by tabs and/or spaces.

-F Do not fork any child *rdist* processes. All clients are updated sequentially.

-f *distfile*
Set the name of the distfile to use to be *distfile* . If *distfile* is specified as ''–'' (dash) then read from standard input (stdin).

-h Follow symbolic links. Copy the file that the link points to rather than the link itself.

-i Ignore unresolved links. *Rdist* will normally try to maintain the link structure of files being transfered and warn the user if all the links cannot be found.

-M *num*
Set the maximum number of simultaneously running child *rdist* processes to *num.* The default is 4.

-m *machine*
Limit which machines are to be updated. Multiple **-m** arguments can be given to limit updates to a subset of the hosts listed the *distfile*.

-N Do not check or update files on target host that reside on NFS filesystems.

-n Print the commands without executing them. This option is useful for debugging *distfile*.

-O Enable check on target host to see if a file resides on a read-only filesystem. If a file does, then no checking or updating of the file is attempted.

-q Quiet mode. Files that are being modified are normally printed on standard output. This option suppresses this.

-R Remove extraneous files. If a directory is being updated, any files that exist on the remote host that do not exist in the master directory are removed. This is useful for maintaining truely identical copies of directories.

-r Do not descend into a directory. Normally *rdist* will recursively check directories. If this option is enabled, then any files listed in the file list in the distfile that are directories are not recrusively scanned. Only the existance, ownership, and mode of the directory are checked.

-s Save files that are updated instead of removing them. Any target file that is updates is first rename from **file** to **file.OLD.**

-t *timeout*
Set the timeout period (in seconds) for waiting for responses from the remote *rdist* server. The default is 900 seconds.

-V Print version information and exit.

-v Verify that the files are up to date on all the hosts. Any files that are out of date will be displayed but no files will be changed nor any mail sent.

-w Whole mode. The whole file name is appended to the destination directory name. Normally, only the last component of a name is used when renaming files. This will preserve the directory structure of the files being copied instead of flattening the directory structure. For example, renaming a list of files such as (dir1/f1 dir2/f2) to dir3 would create files dir3/dir1/f1 and dir3/dir2/f2 instead of dir3/f1 and dir3/f2.

-x Automatically exclude executable files that are in *a.out(5)* format from being checked or updated.

-y Younger mode. Files are normally updated if their *mtime* and *size* (see *stat*(2)) disagree. This option causes *rdist* not to update files that are younger than the master copy. This can be used to prevent newer copies on other hosts from being replaced. A warning message is printed for files which are newer than the master copy.

DISTFILES

The *distfile* contains a sequence of entries that specify the files to be copied, the destination hosts, and what operations to perform to do the updating. Each entry has one of the following formats.

<variable name> '=' <name list>
[label:] <source list> '->' <destination list> <command list>
[label:] <source list> '::' <time_stamp file> <command list>

The first format is used for defining variables. The second format is used for distributing files to other hosts. The third format is used for making lists of files that have been changed since some given date. The *source list* specifies a list of files and/or directories on the local host which are to be used as the master copy for distribution. The *destination list* is the list of hosts to which these files are to be copied. Each file in the source list is added to a list of changes if the file is out of date on the host which is being updated (second format) or the file is newer than the time stamp file (third format).

Labels are optional. They are used to identify a command for partial updates.

Newlines, tabs, and blanks are only used as separators and are otherwise ignored. Comments begin with '#' and end with a newline.

Variables to be expanded begin with '$' followed by one character or a name enclosed in curly braces (see the examples at the end).

The source and destination lists have the following format:

<name>

or

'(' <zero or more names separated by white-space> ')'

The shell meta-characters '[', ']', '{', '}', '*', and '?' are recognized and expanded (on the local host only) in the same way as *csh*(1). They can be escaped with a backslash. The '~' character is also expanded in the same way as *csh* but is expanded separately on the local and destination hosts. When the **-w** option is used with a file name that begins with '~', everything except the home directory is appended to the destination name. File names which do not begin with '/' or '~' use the destination user's home directory as the root directory for the rest of the file name.

The command list consists of zero or more commands of the following format.

'install' <options> opt_dest_name ';'
'notify' <name list> ';'
'except' <name list> ';'
'except_pat' <pattern list>';'
'special' <name list> string ';'
'cmdspecial' <name list> string ';'

The *install* command is used to copy out of date files and/or directories. Each source file is copied to each host in the destination list. Directories are recursively copied in the same way. *Opt_dest_name* is an optional parameter to rename files. If no *install* command appears in the command list or the destination name is not specified, the source file name is used. Directories in the path name will be created if they do not exist on the remote host. To help prevent disasters, a non-empty directory on a target host will never be replaced with a regular file or a symbolic link. However, under the '–R' option a non-empty directory will be removed if the corresponding filename is completely absent on the master host. The options

–N –O –R –b –h –i –q –r –s –v –w –x –y

have the same semantics as options on the command line except they only apply to the files in the source list. The login name used on the destination host is the same as the local host unless the destination name is of the format ''login@host".

The *notify* command is used to mail the list of files updated (and any errors that may have occured) to the listed names. If no '@' appears in the name, the destination host is appended to the name (e.g., name1@host, name2@host, ...).

The *except* command is used to update all of the files in the source list **except** for the files listed in *name list*. This is usually used to copy everything in a directory except certain files.

The *except_pat* command is like the *except* command except that *pattern list* is a list of regular expressions (see *ed*(1) for details). If one of the patterns matches some string within a file name, that file will be ignored. Note that since '\' is a quote character, it must be doubled to become part of the regular expression. Variables are expanded in *pattern list* but not shell file pattern matching characters. To include a '$', it must be escaped with '\'.

The *special* command is used to specify *sh*(1) commands that are to be executed on the remote host after the file in *name list* is updated or installed. If the *name list* is omitted then the shell commands will be executed for every file updated or installed. The shell variable 'FILE' is set to the current filename before executing the commands in *string*. *String* starts and ends with '"' and can cross multiple lines in *distfile*. Multiple commands to the shell should be separated by ';'. Commands are executed in the user's home directory on the host being updated. The *special* command can be used to rebuild private databases, etc. after a program has been updated.

The *cmdspecial* command is similiar to the *special* command, except it is executed only when the entire command is completed instead of after each file is updated. The list of files is placed in the environment variable **$FILES.** Each file name in **$FILES** is seperated by a ':' (semi-colon).

If a hostname ends in a ''+'' (plus sign), then the plus is stripped off and NFS checks are disabled. This is equivilant to disabling the **–N** option just for this one host.

The following is a small example.

```
HOSTS = ( matisse root@arpa)

FILES = ( /bin /lib /usr/bin /usr/games
          /usr/include/{*.h,{stand,sys,vax*,pascal,machine}/*.h}
          /usr/lib /usr/man/man? /usr/ucb /usr/local/rdist )

EXLIB = ( Mail.rc aliases aliases.dir aliases.pag crontab dshrc
          sendmail.cf sendmail.fc sendmail.hf sendmail.st uucp vfont )
```

```
${FILES} -> ${HOSTS}
        install -R ;
        except /usr/lib/${EXLIB} ;
        except /usr/games/lib ;
        special /usr/lib/sendmail "/usr/lib/sendmail -bz" ;

srcs:
/usr/src/bin -> arpa
        except_pat ( \\.o\$ /SCCS\$ ) ;

IMAGEN = (ips dviimp catdvi)

imagen:
/usr/local/${IMAGEN} -> arpa
        install /usr/local/lib ;
        notify ralph ;

${FILES} :: stamp.cory
        notify root@cory ;
```

FILES

distfile - input command file
/tmp/rdist* - temporary file for update lists

SEE ALSO

sh(1), csh(1), stat(2), rcmd(3)

DIAGNOSTICS

BUGS

Source files must reside on the local host where rdist is executed.

Variable expansion only works for name lists; there should be a general macro facility.

Rdist aborts on files which have a negative mtime (before Jan 1, 1970).

NAME

rdistd – remote file distribution server program

SYNOPSIS

rdistd –S [–D]

rdistd –V

DESCRIPTION

Rdistd is the server program for the *rdist* command. It is normally run by *rdist* via *rsh(1).*

The **–S** argument must be specified. The option is required so that *rdistd* is not accidentally started since it normally resides somewhere in a normal user's **$PATH.**

OPTIONS

–D Enable debugging messages. Messages are logged via *syslog(3).*

–V Print version information and exit.

FILES

SEE ALSO

rdist(1), rsh(1)

BUGS

doit: A Network Software Management Tool

Mark Fletcher – SAS Institute Inc.

ABSTRACT

Imagine seven system administrators maintaining nearly 800 HP 9000/700 series workstations, all interconnected using AFS and NFS, and running the SAS System as well as a host of other third party and locally developed software. In February 1991, we realized that in just nine months, we would have to have such an environment in place and would have to be able to keep it that way.

SAS Institute, Inc. made the decision in May 1991 to replace its software development base of 400 Apollo workstations with the newly released HP 9000/700 series workstation. The "conversion" was due to be completed in September 1992.

From the experience of trying to maintain 400 Apollo workstations, we knew we were going to need a tool that would automate and neatly organize the custom software environment of our new systems. We also knew we would be losing the distributed file system (DFS) which originally attracted us to the Apollo. There were also several in house tools on the Apollo which made software maintenance easier than on the HPs. Nevertheless, keeping software up to date on 400 Apollos was quite a chore. We knew that the HP workstation count would likely double within a year and that we couldn't keep doing things the way we always had. We needed a tool that allowed a client (a typical workstation on the network) to, in a "self discovering" manner, find out about software updates, perform the updates and log its activities - all without human intervention. We also wanted the system to be table driven, so we could view the table and see at a glance what software should be installed on each machine. In addition, we wanted the tool to be platform independent so that Suns, DECs, HPs, etc. could be maintained with the same tools. Finally, the nature of our software development base required certain machines to be configured differently from others. This required our software update tools to be able to determine the *class* of a machine and install customized kernels, device drivers, etc. appropriately.

The tool we implemented was affectionately named **doit** in the spirit of Captain Jean Luc Picard's famous command "Make it so" which became our theme during the nine month conversion preparation effort.

The initial version of **doit** consisted of a number of shell scripts and C programs which were downloaded to a client from a special machine known as a *librarian*. This librarian contained the software to be downloaded as well as the table (called the *action* file) which directed the update. The librarian was an HP 9000/750 file server which was later replicated to 19 similar servers just prior to the installation of the largest wave of new workstations (about 300). We had HP's Integration Center add a call to **doit** in */etc/rc* before shipment so that **doit** would run every time the machine booted. A revision number (called the **doit number**) was kept on each machine's disk so **doit** could determine what software (if any) was needed to bring the machine up to date; thus the automated nature of the update process.

After some 800 installs, **doit** was rewritten entirely in C as a TCP/IP client-server model. A tenfold increase in speed was realized and we closed some rather gaping holes in network security as well.

The Apollo Days

At one point before we converted our software development base to the HP workstation, we were administering over 400 Apollos. Apollo's Domain File System made the task easier; still, much administration was done by hand. It was often difficult to determine what customizations a particular workstation had received and how up-to-date things were. Operating system updates were done using Apollo's native tools. **SAS** customizations were done on a case by case basis. We tried to discourage individual machine customizations because it was so hard to keep track of them. However, we had to customize some machines which made it very hard to tell what state machines were in at any given time.

During planning for the HP conversion, we realized we were going to have to develop a good software installation and maintenance system to meet the challenges of the new computer platform. We determined that the major issues were:

- Many of the Apollo network tools we relied on so heavily would not be available on the HP.
- We had been told that the machine count would easily double from the 400 Apollos we were currently administering. We couldn't keep doing so many things by hand.
- We wanted each machine to maintain a *revision number* that reflected how many of the customizations had been installed. With this we could tell at a glance what state the machine was in.
- We wanted to maintain a "road map" of customizations for every machine in our network so the update process could be automated. We also wanted this configuration file (we later named it an *action* file) to be maintained in a single location so it would be easy to manage.
- Since we knew our users were going to require specialized software (custom kernels, special compilers, editors, etc), we needed a system that could manage classes of hosts; i.e., one class of machines used by the technical writers, another by the C Compiler Development group, one for the Computer Graphics Development group, etc.
- For ease of maintenance, we wanted to have all the software updates kept in a single (and perhaps replicated) server.
- We wanted our system to be as generic as possible. We wanted to stay away from HP specific tools so other non-HP machines could also be maintained with the same software.
- Automated operation was a must. We needed client machines to realize they needed updates and to perform the updates automatically. This "self discovery" would take place at boot time. The nature of an update would be determined by the machine's revision number, the host class of the machine, and instructions found in the action file that would be appropriate for that client. As long as a machine rebooted occasionally, that machine's software would always be current.

The Design of doit

doit is designed around a TCP client-server model. The client attaches to the server via TCP sockets. The server is started by **inetd**. Figure 1 shows the sockets and files used by the client (**doit**) and the server (**doitd**) during an update session.

We place a call to the **doit** client binary in every machine's */etc/rc* file so **doit** will run every time a machine boots. To attempt to distribute the load on a **doit** server, we replicate the server software across several servers.

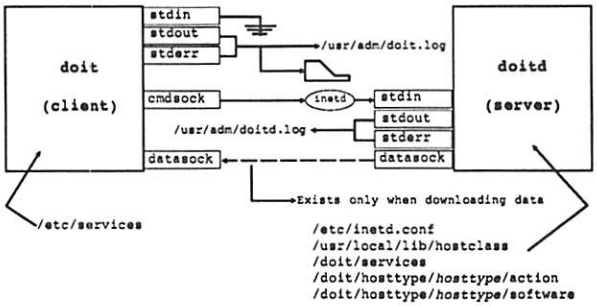


Figure 1: doit I/O connections and files

When **doit** starts, it connects to a special *librarian* machine. The purpose of this machine is to provide a list of possible **doit** servers that the client can choose from so as to balance the load across servers. The librarian maintains a file, */doit/servers*, which contains a list of all machines which are participating **doit** servers. It is typical for the librarian to also be one of the **doit** servers because it is the **doit** program that performs the librarian function via a GETSERVERS request. The only extra software a **doit** server needs to be a librarian is the */doit/servers* file. To prevent a "single point of failure" problem, we replicate the librarian across three machines using the following aliases which we place in **named**.

```
librarian0
librarian1
librarian2
```

The number of librarian replicas is a constant compiled into the **doit** binary. Because the amount of information the librarian supplies is quite small, having more than three librarians is rarely necessary.

The **doit** program receives the list of **doit** servers and makes a random[1] choice of a server.

doit reads the */etc/services* file to obtain the port number to connect to on the server it has selected. The server must have a corresponding entry in its */etc/inetd.conf* file that will start **doitd**. To keep things simple, we distribute the same */etc/inetd.conf* file to all machines and then only distribute **doitd** to those we want to actually be **doit** servers; see Figure 2.

The client program redirects its standard output and standard error to a log file; typically */usr/adm/doit.log*. This assures that everything that goes to the screen while **doit** is running will also be

[1]The decision is somewhat random because it is biased by the IP address of the server. **doit** tries to minimize network hops in its selection of a **doit** server. Also, **doit** will skip a server if it appears to be down.

saved in this log. The name */usr/adm/doit.log* is compiled into **doit** and is specified in an include file. Similarly, **doitd** redirects stdout and stderr to */usr/adm/doitd.log*. Just to be safe, **doit**'s standard input is redirected to */dev/null* since it is never used. **doitd** leaves its standard input connected to the socket created by inetd. The client will create a TCP socket (*cmdsock*) that will attach to **doitd**'s standard input. This socket is then used to submit commands to **doitd**. These connections are illustrated in Figure 1. The possible commands **doit** can request of **doitd** are:

QUIT Close down the session and terminate.

GETSERVERS Provide a client with a list of available **doit** servers. This is the command **doit** receives when it is behaving as a librarian.

GETACTION Download an *action* file (a set of software update instructions specific for that host based on its revision number and host class). The action file is described in the next section.

DOWNLOAD Download a file or directory from the server's update software area.

The action file (the update "road map") and update software directory are kept on the server; typically in the directory /doit/hosttype/*hosttype*. *hosttype* is a vendor name like HP700, SUN4, DEC5000 and is compiled into the client's copy of the **doit** binary. Since each vendor type requires its own binary anyway, it is not inconvenient to include this string in the source for **doit**. For our systems, this path name is */doit/hosttype/HP700*.

Another socket, *datasock*, is created each time a GETSERVERS, GETACTION or DOWNLOAD command is issued. This socket is used for the data being downloaded. The socket is then dissolved after the download is complete. The client creates the socket first, then sends the port number of *datasock* via the command socket to the server. The server can then complete the connection.

The Action File

As stated earlier, the *action* file is **doit**'s "road map". Everything **doit** does is strictly dictated by the action file. The server, after establishing a connection with the client, extracts certain lines from the action file and downloads them to the client. The selected lines are those necessary to bring the client up to the latest set of updates.

Action File Format

```
Action-type Hostclass-expression \
Revision-level action-specific-fields
```

Action Type

doit performs three types of actions

1. Add software (download)
2. Delete software
3. Run a shell command

Sometimes it is necessary to reboot the machine following an update, a series of updates, or when all updates have been performed. For example, if you have just installed a new kernel with an "add software" command and then wanted to run some commands that depended on your being up on that kernel, it would be necessary to reboot the machine after the kernel is installed. In another case, you might find that you could finish all updates before you reboot. Either way, a method for rebooting within **doit** becomes necessary. The following syntax shows all the variations available.

a Add (download) a file or directory from the server to the client.

A Same as **a** but causes a reboot at the end of the current *revision level* (described in a later section).

A+ Same as **a** but causes a reboot after ALL actions are performed.

NOTE: The download is done using tar. All source permissions and ownership are preserved.

d,D,D+ Delete a file or directory.

r,R,R+ Run a shell command. (Uses Bourne Shell)

Hostclass Expressions

A host class[2] classifies a machine to be of a certain type or to have a certain function. Physically, a host class is a file containing a list of all machines that make up that class. For example, we have one host class named "HP750" that contains the names of all our HP file server machines. The "HP720" class contains all the other (regular) workstations. We have lots of other host classes like "HP720.PUB" for the HPs in our Publications

[2]Host classes are groups of host names which are related in some way. The program, **expandhosts**, which evaluates expressions involving host classes was written by the Microelectronics Center of North Carolina (MCNC) and was adopted by SAS for various uses.

```
$ grep doit /etc/services /etc/inetd.conf
/etc/services:doitd     5025/tcp
/etc/inetd.conf: \
doitd stream tcp nowait root /usr/local/etc/doitd doitd
```

Figure 2: Startup code

Department, "HP720.CCD" containing a list of all the machines in the C Compiler Development department.

The host class files are stored on the server in */usr/local/lib/hostclass*.

A host class expression is based on set logic. The standard set operators are supported. The symbols used are:

% Intersection

+ Union

- Difference

( ) expression grouping

name The name of a host

$*name* The name of a host class

NOTE: There are no precedence rules. Expressions are processed from left to right.

In addition to evaluating the revision level to determine if a certain action should be executed, the **doit** server evaluates the host class expression of the action line. If the host class expression **intersected** with the client's host name returns the client name, then the line is determined to be appropriate for the client.

For example, if an action line contains the expression

```
$hp720-$hp720.pub
```

(which means "all regular HP workstations except those in the Publications Dept.") then **doitd** would evaluate

```
$hp720-$hp720.pub%clientname
```

If the expression yields the client name then we know the client is an HP720 that is not in the Publications Department.

Revision Level

The client maintains a revision level number stored in */doitlevel*. The number is of the form:

dd.dd

When the server downloads the action lines to the client, only lines with a revision level higher than the client's */doitlevel* value will be sent. As the client processes the lines, its revision level is updated to match that of the line being executed. This prevents lines from being processed more than once. After all lines have been processed, the highest revision level that appeared in the action file is written to */doitlevel*. The next time **doit** runs, none of these lines will be downloaded because of the client's revision level. Thus, **doit** will exit without doing anything.

One of the things the server does while preparing to download the action lines is to sort the action file by revision level. That is, all the "1.00" actions are grouped together, as are "1.01" lines, etc. **doit** does not sort the lines within a revision level; they are just grouped together in the order they appear in the action file.

The "@" and "*" Revision Levels

There are two special revision levels that are used when you want an action to be performed every time **doit** runs. The server will always download these lines regardless of the client's revision level. "@" lines always sort to the top of the action list and are therefore processed before any other line. "*" lines always sort to the bottom and are executed after all other actions.

Action Specific Fields

The remainder of the action line depends on the command type.

Add Line

```
a hostclass-expression revision-level \
source-path dest-path
```

source-path The path name on the server. This is the file/directory to be downloaded. If the path name is relative, it is relative to the top of where the source software directory resides on the server; e.g.,

```
/doit/hostclass/HP700/software.
```

dest-path The target file/directory on the client. This must be an absolute path.

When the **add** command executes, the destination path name is first deleted. If the path name is busy and won't delete, it is renamed to a file name beginning with "#". As a convenience, **doit** will look for an old "#" file and try to delete it.

Delete Line

```
d hostclass-exprrevision-level \
                dest-path
```

dest-path is deleted from the client's file system. The same process is followed to deal with busy files as in the case with the **add** command. This command requires no interaction with the server.

Run Line

```
r hostclass-exprrevision-level \
                command
```

A Bourne Shell is spawned to run command on the client. Any shell syntax can be used within the command including file redirection and pipes. This command requires no interaction with the server.

An Example doit Session

In this example, let's assume we have five HP750 servers which provide **doit** services:

- borg
- connors
- macenroe
- lendl
- graf

We also assume **named** contains the following aliases:

- borg: librarian0
- connors: librarian1
- philly: librarian2

The purpose of a librarian machine is to supply **doit** server names to clients. Therefore, a librarian machine maintains the file */doit/servers*. For these example librarians, the contents of */doit/servers* is:

```
borg
connors
macenroe
lendl
graf
```

Note that it is perfectly ok for a machine to double as a librarian and a **doit** server.

doit will first try to contact *librarian0* (**borg**) to get the list of **doit** servers. If *librarian0* does not respond, **doit** will then try *librarian1*. On no response from *librarian1*, **doit** will try *librarian2*. **doit** gives up with an error message if there is still no response.

Five host classes exist in our example. The host class files are stored in */usr/local/lib/hostclass*:

hp750 All HP750 server machines

hp720 All HP720 "regular" workstations

hp720.pub All HP workstations in the Publications Department

hp720.qa All HP workstations in Quality Assurance

hp720.market All HP workstations in the Marketing Department

The action file is stored in the file */doit/hosttype/HP700/action* on the server machines and contains all actions for every machine in the network. Figure 3 shows an example *action* file for the HP700.

The */doit/hosttype/HP700/software* directory would contain all software used by the action file.

```
borg$ ls -FC /doit/hosttype/HP700/software
```

```
# Action file for the HP700 series machines
# Revision level 1.00
d $hp750+$hp720 1.00 /usr/lib/cron/cron.allow
A $hp750 1.00 hp-ux.750 /hp-ux
A $hp720 1.00 hp-ux.720 /hp-ux
#
# Revision level 1.01
a $hp750+$hp720 1.01 etc/rc /etc/rc
a $hp750+$hp720-tester 1.01 tmp/enable_nfs /tmp/enable_nfs
r $hp750+$hp720-tester 1.01 /tmp/enable_nfs
d $hp750+$hp720-tester 1.01 /tmp/enable_nfs
a $hp750+$hp720-tester 1.01 etc/automounter /etc/automounter
a $hp750+$hp720-tester 1.01 tmp/enable_automounter /tmp/enable_automounter
R+ $hp750+$hp720-tester 1.01 /tmp/enable_automounter
d $hp750+$hp720-tester 1.01 /tmp/enable_automounter
#
# Revision level 1.02
a $hp750+$hp720 1.02 etc/shutdown.allow /etc/shutdown.allow
#
# Revision level 2.00
a $hp720.pub 2.00 local/bin/ileaf /usr/local/bin/ileaf
a $hp720.pub 2.00 local/lib/ileaf /usr/local/lib/ileaf
a $hp750+$hp720-$hp720.pub 2.00 local/bin/wp /usr/local/bin/wp
a $hp750+$hp720-$hp720.pub 2.00 local/lib/wp /usr/local/lib/wp
a $hp720-($hp720.pub+$hp720.market) 2.00 usr/lib/xpass.ansi /usr/lib/xpass.ansi
a $hp720-($hp720.pub+$hp720.market) 2.00 bin/cc.ansi /bin/cc
#
# Actions to be run only after everything else is done
a $hp720-($hp720.pub+$hp720.market) * tmp/ansi_c_test /tmp/ansi_c_test
r $hp720-($hp720.pub+$hp720.market) * /tmp/ansi_c_test
d $hp720-($hp720.pub+$hp720.market) * /tmp/ansi_c_test
```

Figure 3: HP700 *action* file

```
etc/                hp-ux.750*      tmp/
hp-ux.720*          local/usr/
```

The other necessary files the **doit** server must contain are the **doit** server binary and the **inetd** config file.

```
borg$ ls -l /usr/local/etc/doitd
-rwxr--r--   1 root      bin         79546 \
Aug 26 16:35 /usr/local/etc/doitd

borg$ grep doitd /etc/inetd.conf
doitd stream  tcp      nowait  root\
/usr/local/etc/doitd doitd
```

Next, we will assume that there is a certain client machine named *frost* which is used in the Publications Department. The machine is classified as an HP720 and is one of the Publications workstations.

```
borg$ grep frost /usr/local/lib/hostclass/*
hp720:frost
hp720.pub:frost
```

frost has a **doit** revision level of 1.00.

```
frost$ cat /doitlevel
1.00
```

In order for the client to run **doit**, it must have a */etc/services* entry for **doitd**, the client binary, and a call to **doit** in */etc/rc*.

```
frost$ grep doit /etc/services
doitd    5026/tcp  # doit server

frost$ ls -l /usr/local/bin/doit
-rwxr--r--    1 root      bin\
83930 Aug 27 10:53 doit

frost$ cat /etc/rc
        :
        :
if [ -x /usr/local/bin/doit ]
         /usr/local/bin/doit
fi
        :
        :
```

Client-server Dialogue

What follows is the dialogue between the client *frost*, the librarian *borg*, and the **doit** server *connors*.

1. *frost* is rebooted
2. *frost* runs **doit** from the *rc* file.
3. **doit** attempts to connect with **doitd** on *librarian0* (*borg*). **doit** gets the port number from */etc/services*. Assume the connection succeeds.
4. **doitd** reads its */doit/servers* file and sends **doit**:
   - o borg
   - o connors
   - o macenroe
   - o lendle
   - o graf
5. **doit** randomly[3] picks a server from the list. We will assume **doit** chooses *connors*.
6. **doit** then connects to **doitd** on *connors*.
7. **doit** sends **doitd** the client's host name and revision level (frost,1.00).
8. **doitd** builds a list of actions for *frost* based on the host class and revision level. *frost* will receive the actions shown in Figure 4. **doit** saves these action lines into the file */tmp/action*
9. **doit** then reads each line of */tmp/action* and performs the appropriate action. The action lines are processed in sequence. Note that the host class is no longer needed but it is easier for **doitd** to send the entire line than to parse it out.

[3]The selection is biased based on the number of router hops and, assuming no changes in the network topology, a given client will always choose the same server because the same "seed" is used every time. We felt it was a bad idea to have clients truly picking servers at random when what we really wanted to accomplish was the distribution of load across servers. The consistent picking of the same server by a specific client helps us reproduce a malfunction in the **doit** system should one occur.

```
a $hp750+$hp720 1.01 etc/rc /etc/rc
a $hp750+$hp720-tester 1.01 tmp/enable_nfs /tmp/enable_nfs
r $hp750+$hp720-tester 1.01 /tmp/enable_nfs
d $hp750+$hp720-tester 1.01 /tmp/enable_nfs
a $hp750+$hp720-tester 1.01 etc/automounter /etc/automounter
a $hp750+$hp720-tester 1.01 tmp/enable_automounter /tmp/enable_automounter
R+ $hp750+$hp720-tester 1.01 /tmp/enable_automounter
d $hp750+$hp720-tester 1.01 /tmp/enable_automounter
a $hp750+$hp720 1.02 etc/shutdown.allow /etc/shutdown.allow
a $hp720.pub 2.00 local/bin/ileaf /usr/local/bin/ileaf
a $hp720.pub 2.00 local/lib/ileaf /usr/local/lib/ileaf
```

Figure 4: Customized actions for *frost*

Line 1

```
a $hp750+$hp720 1.01 etc/rc /etc/rc
```

First, **doit** checks to see if an old "busy" copy of */etc/rc* was created in a previous run of **doit**. The file would be named */etc/#rc*. It deletes it if it exists. Then **doit** tries to delete */etc/rc*. If the file is busy and won't delete, it is renamed */etc/#rc*.

Next, **doit** sends a "download" request and the string *etc/rc* to **doitd**. **doitd** recognizes the path name to be relative and changes it to */doit/hosttype/HP700/software/etc/rc*. The file is then **tar**ed to *frost* where it is copied into */etc/rc*. All permissions and ownership are preserved.

After the download completes and **doit** verifies that the new file is there, **doit** writes a number corresponding to the line number in */tmp/action* of this command to */doitlevel*. The most recently completed revision level is also saved. At this point, the contents of */doitlevel* would contain:

```
Line 1 = Last line completed in /tmp/action
Completed level = 0.0
```

Storing this information allows **doit** to pick up where it left off in the event of a reboot while **doit** is running. This reboot may occur accidentally (power failure) or from a reboot action command type (A, D, or R). When **doit** is first executed, it knows it was interrupted if */tmp/action* exists and */doitlevel* contains information other than a revision level.

Lines 2-4

```
a $hp750+$hp720-tester 1.01 tmp/enable_nfs \
                            tmp/enable_nfs
r $hp750+$hp720-tester 1.01 /tmp/enable_nfs
d $hp750+$hp720-tester 1.01 /tmp/enable_nfs
```

enable_nfs is a shell script that takes all the necessary steps to get NFS up and running on the client. These three lines illustrate the downloading of a special purpose program to the client, running that program, and then deleting it when it is done.

Lines 5-8

```
a $hp750+$hp720-tester 1.01 \
    etc/automounter  /etc/automounter
a $hp750+$hp720-tester 1.01 \
  tmp/enable_automounter  tmp/enable_automounter
R+ $hp750+$hp720-tester 1.01 /tmp/enable_automounter
d $hp750+$hp720-tester 1.01 /tmp/enable_automounter
```

These commands install the NFS Automounter on the client. Line 5 installs the binary itself. Lines 6-8 are similar to lines 2-4. A shell script is downloaded, run, and deleted. One difference here is the **R+** which means that the machine is to reboot after all action lines have been processed. The reason we do this is because the machine must be rebooted after installing the automounter.

Line 9

```
a $hp750+$hp720 1.02 etc/shutdown.allow \
                     /etc/shutdown.allow
```

This line installs a *shutdown.allow* customized for the HP750's and HP720's.

Line 10

```
a $hp720.pub 2.00 local/bin/ileaf /usr/local/bin/ileaf
```

This line installs the Interleaf Desktop Publishing binary on the Publications machines.

After this line is done, the client has been updated through revision level 1.02. Therefore, the */doitlevel* file will look like this:

```
Line 10 = Last line completed in /tmp/action
Completed level = 1.02
```

Line 11

```
a $hp720.pub 2.00 local/lib/ileaf /usr/local/lib/ileaf
```

Installs the Interleaf libraries.

frost is now fully updated.

10. The contents of */doitlevel* is replaced with the revision level number; in this case, 2.00.

```
frost$ cat /doitlevel
2.00
```

11. */tmp/action* is deleted.
12. **doit** sends a QUIT request to **doitd** and terminates.
13. **doit** issues a shut down command because of the **R+** in line 7.

The Conversion

doit's debut was during the replacement of the 400 Apollo workstations with HPs. We refer to this as the Conversion.

Most of the conversion occurred during two weekends. The first wave of about 100 machines marked the first real test for **doit**. In fact, one of our administrators worked until about 2:30 a.m. the morning of the conversion putting the action file together. **doit** performed admirably.

The next phase of the conversion involved about 500 machines. **doit** didn't fair so well this time. At the time, we were not replicating our **doit** server; we had only one. It didn't seem to be a problem while installing the 100 machines the weekend before, but when 500 machines hit the single **doit** server, everything ground to a halt. We sent all our helpers out for a long lunch break (about three hours) while we frantically added the server replication code to **doit**. We set up 25 HP750s as **doit** servers and had clients randomly choose between them. **doit** did pretty well after that.

doit consisted of a set of shell scripts until about six months after the conversion when it was rewritten in C. The biggest problems with the shell version was it was extremely slow at building an action file and there were serious security problems.

Even when a machine was up-to-date, **doit** spent between three to five minutes plowing through the action file only to realize that there was nothing to do. In the new version of **doit**, the building of the action file was moved from the client machine to the server. Since the host classes reside on the server, the old **doit** spent most of its time passing remote shell commands back and forth. The new **doit** builds the entire action file on the server and then downloads the entire list at one time. This reduces the number of times the client and server must communicate to one. The host class evaluation software had a number of problems of its own so we decided to rewrite it too. After all this, the worst case action file build time went from five minutes to five seconds! The fact that **doit** is no longer a shell program has contributed to the dramatic performance improvement too.

A bigger problem than speed was the gaping security holes. Since **doit** used **rcp** and **remsh** throughout, the server's */.rhosts* file had to be wide open (i.e., contained "+"). This meant that anyone with root access anywhere on the network also had root access to the servers. The new **doit** closes these holes because it uses TCP ports to communicate.

Future Enhancements

To date, **doit** has only been used to update HP700 machines. Although support for other vendors was designed in from the beginning, it has never been tested. This should not pose a significant problem however since the only outside program employed by **doit** is **tar**. Great care was taken to code **doit** in a portable manner.

We wanted to use **hesiod** instead of having to maintain *librarian[0-n]* aliases in **named**. We opted for the way we did it because we had not ported **hesiod** to the HP system at the time.

Author Information

Mark Fletcher is a system administrator for the Unix support group at SAS Institute Inc. Reach him via U.S. Mail at SAS Institute Inc.; SAS Campus Drive; Cary NC 27513. Reach him electronically at mark@unx.sas.com.

PITS: A Request Management System

David Koblas – Independent Consultant
Paul M. Moriarty – cisco Systems, Inc.

ABSTRACT

A key component of a successful computer services group is its ability to provide timely responses to requests from the user community. As the computing environment continues to grow, increasingly greater amounts of the group's time are diverted from servicing requests to the overhead of managing incoming and outstanding requests. Reducing this overhead frees up more time to do more productive work.

This paper describes the design and implementation of the Problem and Information Tracking System (PITS) for the Engineering Computer Services group at MIPS Computer Systems, Inc. The system takes a comprehensive approach to request management by providing a set of utilities for request submission, assignment and closure as well as acting as the interface to a relational database back end.

Introduction

Engineering Computer Services (ECS) was chartered with providing support to the engineering computing environment at MIPS. This environment eventually consisted of about 250 users, 125 servers, 200 dataless workstations and 50 Xterminals.

The first attempt at request management at MIPS started with an e-mail alias *sysadmin* which was the single point of contact for requests from the user community. Users would send their requests via e-mail to this alias and the ECS staff would periodically check the mailbox and work on outstanding requests. This scheme was designed to ensure that requests would be serviced by ECS regardless of who was on site on any particular day.

Problems with this scheme included:

- The UNIX mailbox format prevented more than one person from working on the list of outstanding requests at a time.
- It was difficult for staff members to determine the status of a request (i.e., new, assigned, open or closed).
- Searching for similar requests was done using primitive tools (**grep(1)**).
- Gathering useful management statistics beyond number of messages per day was impossible.
- Users could not determine the status of their request without contacting an ECS staff member who often ended up simply asking all of the other staff members if they were working on the requests until the correct person was identified.
- Users could not determine if their request had already been made by another user as they did not have permission to read the mailbox.
- Users had no way of knowing how many requests were outstanding.

When the number of requests per day approached 35, it became a full time job for one administrator to manage the sysadmin mailbox. This person would service the requests that could be completed quickly and would assign the other, more complex requests to other ECS staff members.

Eventually, this scheme no longer worked. The administrator managing the mailbox would eventually fall behind in processing the queue of incoming requests and, as the backlog of requests grew, the users would become frustrated at not having their request serviced in a timely manner. Many complained that it sometimes took hours before they even received an acknowledgment that their request had been received. The ECS staff decided to try to solve the problem of request management. We looked at packages that were freely redistributable as well as commercial packages.

The **queuemh** system was the only freely redistributable package that we found. **Queuemh** was nice in that it had the capability to work on a request for a while, add some status regarding what was done and put the request aside but keep the original request open. It lacked the ability to directly assign each request to an administrator in a manner that also permitted a way of easily viewing all open requests.

Several commercial packages were investigated. Most employed some commercial database as a back end. The database solved a lot of the problems surrounding searching requests as well as collating open and closed requests. However, the commercial systems had drawbacks that were unacceptable to us. It seemed that most were designed for more traditional customer service applications where each person managing requests dealt with a small number of very detailed requests each day. Our system needed to be able to handle many requests containing very small amounts of detail by comparison.

Design Goals

We decided to write a set of utilities to interface with a back end commercial database. The design of the Problem and Information Tracking System (PITS) was started with the following goals:

- Users should be able to submit requests from their terminal, similar to sending e-mail.
- The system needed a text based interface. While most users had some sort of graphics display terminal on their desktop, we envisioned that users might also wish to submit requests when working from a plain ascii terminal (e.g., when working from home).
- Submitting requests should be as quick and easy as possible. If it became difficult or awkward to submit requests, the user community would not accept the new scheme.
- The submitter should be automatically informed of all changes in the status of the request via e-mail.
- An SQL-based database for ease of statistics gathering, report generation and searching through requests.
- Users should be able to make arbitrary queries into the database.
- The systems administrators should be able to use a familiar interface when interacting with the database to manage requests.

Request Life Cycle

The life cycle of a request within PITS is simple. There are four phases: generation, assignment, in-process and closure.

Request Generation

The user community makes requests through one of three interfaces, ascii text or X Windows request screen and e-mail. All of these interfaces gather five items of information:

- User name.
- Affected host.
- A short summary description.
- An arbitrarily long description.
- An optional list of other users to keep updated about the request's status.

An example of the ascii submission screen is shown in Figure 1.

Before a request actually enters the database, some validation is performed by the submission utilities against the fields. The existence of a short description is verified and the host and user name fields are checked to ensure there is no intra-word spacing. The submitter's user id and submitting host name are recorded for future reference (users occasionally target the wrong machine as having a problem, particularly when the problem is network related). If any of the data in these fields is deemed invalid, the submitter is prompted to correct the field before the request can be submitted. If the request is submitted via e-mail, the user name, affected host and short description fields are extracted from the mail header. The Subject: line becomes the short description. If the subject line is missing, a "no subject" entry is made in the short description field.

```
Engineering Computer Services Problem Report


Affected Host: ecstasy___                          Submitter Name: koblas__

Short Description:
  Hardware lab systems_____________________________________________________

Edit Extended Description (y/n): _

Extended Description:
  Can someone please reroute the power cords to the systems housed  [...]
  Hardware lab?  I plan to move some of the benches to SGI this Fri [...]
  of the power cords to systems like minnie and monie are routed th [...]
  of one of the benches.    Thanks,


Cc: pmm_____________________________

Submit this problem (y/n/q): _
```

Figure 1: Ascii submission screen

Once a request has been entered into the database, an e-mail acknowledgment is sent to the submitter and anyone on the carbon copy list. The request now appears on the dispatcher's request tracking screen or request box.

Request Assignment

The administrator's view of PITS is that of a request box. All newly generated requests are sent to a request box that is managed by a dispatcher. The dispatcher reviews and either services the request directly or assigns the request to an administrator for action. A (Note: Request boxes are really an abstraction. In reality, the request box for person 'a' is simply a display of all requests in the database assigned to person 'a'. The default request box for all incoming, unassigned requests at MIPS is 'ecs'.)

The request is then "sent" to the request box of the administrator to which it has been assigned. E-mail notification of the assignment is automatically sent to the submitter of the request.

Request boxes use an interface that is very similar to the one used by the **Elm** mail user agent. We decided to use this interface because it was one that was familiar to most of the systems administrators in ECS. An example of the request box interface is shown in Figure 2. A request is examined by moving the arrow to the desired request id and hitting the return key. All of the information about the request is then displayed. An example of a full description is shown in Figure 3.

Using PITS the administrator can easily take any of the following actions:

- Re-assign the request to another administrator. If, after some investigation, an administrator decides that the request would best be handled by another administrator, the first administrator can reassign the request after entering any informational comments regarding the request. At this point, the request 'moves" to the request box of the new administrator. At the same time, e-mail notification of the reassignment along with the informational comments is automatically sent to the requester
- Append more information to the request description. Useful if a request will be worked on for a while and then set aside to be worked on later.
- Mark the request inactive. Some requests may take a long time to service (e.g., adding some requested enhancement to a piece of supported software) and we wished to note this somehow in the database. We accomplish this by marking the request as "inactive". While somewhat of a misnomer, it serves to denote those requests which will take longer periods of time to service.

```
                    Current box is "incoming" with 134 records

      PID Date   Assign  Pri Short Description

  I  3320 06-MAR jc          can ping it from tajmahal but cannot run anyt
->   3321 06-MAR timw        fp chip on heineken
     3322 06-MAR timw        egl1 is sick, tx'es even when ifconfig'ed off
     3323 06-MAR timw        latest kernels incorrectly installed in ECS m
     3329 07-MAR jpatton     Re: opshell (fwd)
  I  3332 08-MAR jpatton     Term, Randy Zach
     3346  09:02 jpatton     please move me from hal to igloo (with the re
     3364  11:56 gentile     ncd down
  N  3368  13:57 ecs         new hire - Earl Devenport (fwd)

        !=shell, ?=help, <n>=set current to n, /=search pattern, n)ew
     A)ppend, a)assign, C)lose, d)uplicate, I)nactive, m)ail, P)riority
           c)hange screen, s)ubmit, t)ag, l)imit, p)rint, q)uit

   Command:


                              A problem has changed
```

Figure 2: Request box interface

- Request information from the submitting user via e-mail. If there is insufficient information present in the request to act upon it, the administrator can use the request box interface to send email asking for additional information. The outgoing mail references the request id number in the Subject: line and has a special Reply-To: address inserted so that when the user responds, the information in the response is automatically appended to the outstanding request. The request can optionally be marked inactive at this point and when the user responds the request will automatically become active again.
- Mark the request low, normal or high priority.
- Resolve the request. Actually act upon and close the request.

PITS also permits the administrator to perform some basic request organization:

- Tagging or grouping a collection of similar requests together, such that any action that is taken affects the group of requests rather than an individual report. Useful when several users make the same request (e.g., requests to investigate a machine that appears to be down).
- Limiting the displayed requests list to a set that contains a specific substring.
- Quick searching and jumping around the request box, via string searches, numeric request id or by jumping to the first 'new' request.

```
Problem ID: 3364

   Host: ncddee                              Submit Host: widget
   User: deedee (Dee Dee France)             Submit User: gentile
   Cc  : ,jes,gentile

   Status      : closed                 Open Date : 09-MAR-92 11:56:48
   Assigned To: closed                  Close Date: 09-MAR-92 14:06:31
   Priority    :                        Closed By : gentile
                                     Problem Type : ws.down

--------------------------------- Description ---------------------------------

Subject: ncd down
--

---------------------------- Tracking Information ----------------------------


Date: 09-MAR-92 12:00:44      Assigned To: gentile          By: gentile
Date: 09-MAR-92 14:06:31      Assigned To: closed           By: gentile
Analysis ----

The ncd was attempting to arp its IP address
rather than get it from its nvram.
This selection was changed by the effect of
a bug in the rather old (1.1?) ncd boot proms.
(Thanks, John! I never knew that.)

I don't know why the boot would fail to load (file not found)
when batman's /tftpboot link was temporarily removed.

-Don

-----


End of report #3364, Command:
```

Figure 3: Full Description

Requests in-process

Users and administrators are provided with daily e-mail summary of the status of their outstanding requests and who is working on them. Inactive requests are summarized weekly.

A utility called *showit* was written to allow a simple set of queries to be made into the database. These queries allow the user to view the entire request reports and information contained in the database for any requests.

Request closure

Once a request has been serviced, it is closed by the systems administrator. The administrator is asked to classify the request into one of several problem categories (e.g., machine down, network problem, set up new account, etc...) and to enter some comments about what was done to service the request. When the request is closed, e-mail is automatically sent to the submitter, notifying them that the request has been serviced.

Future work

What follows is a list of additional features that would be helpful for future versions of PITS. This list is an outgrowth of the combined experiences of many PITS users.

- Due dates. We would like to provide the submitter with some way of noting when a request needs to be completed,
- Clean interface for status changes. At present, the request box will beep and a message will appear that says "a request's status has changed" but it does not note which request has actually changed. It can be difficult for an administrator with many outstanding requests to figure out which one has changed.
- Priority should be simplified/easier to use. We still haven't figured out a good way to use the priority field. While it seems like a good concept, most administrators are adept at determining what is a high priority and what isn't. As a result, this field tends not to be used.
- Some way to glue requests together, stronger than tagging them. Tagging is useful only while the request box is open. If the administrator exits the request box utility without acting on the tagged requests, the tag is lost.
- Key word search. The is no way to search for key words in the entire body of open and closed requests. We envision that such a tool would be a boon for new users and administrators as they could search this body of knowledge prior to submitting a request or asking others for help.
- Referencing other requests. Sometimes requests are related, but not identical enough that they can be tagged together. There is no way to directly reference other requests other than mentioning them in the comments section.
- Editable cc list. Sometimes people who have been cc'd on the initial request do not wish to receive e-mail about the request's status. At present, there is no way inside of PITS to edit them out of the cc line. This must be done by interacting with the database directly.

Conclusions

MIPS has been running PITS for the past year with only a very limited set of changes to the initial design. The system is currently handling over 70 requests a day. Response from the user community about PITS has been quite positive since the users are better able to follow their requests.

The administrator handling the dispatch function has more time to actually work on requests. Since e-mail acknowledgments are now automatically generated, each request no longer requires a manually entered response from the dispatcher.

System administrators have found PITS to be extremely helpful in keeping track of their outstanding requests. The daily and weekly status messages keep help requests from being inadvertently forgotten.

Reports can be generated for management detailing many aspects of how requests are processed. It is easy to obtain reports using the database's report generation package that show the average time from request generation to closure, a listing of the top types of requests, a listing of the top requesters as well as a list of all outstanding requests.

Author Information

Paul M. Moriarty is the Systems Administration Manager for cisco Systems, Inc. where he leads the group responsible for care and feeding of the engineering computing environment. Prior to that, Paul was the Sr. Systems Administrator at MIPS Computer Systems, Inc where he fiddled with news, email and site security. Paul is a founding and present member of the board of directors Bay-LISA, a San Francisco Bay Area user's group for systems administrators of large sites. Paul is also a member of the interim board of directors of SAGE, as well as the coordinator of the SAGE certification working group. Contact Paul via US Mail at cisco Systems, Inc; 1525 O'Brien Drive; Menlo Park, CA 94025 or via e-mail at pmm@cisco.com.

David Koblas is an independent consultant who has assisted with application development at many large Silicon Valley companies. In addition to his consultant business, Dave is busily at work on version 2.0 of **Xpaint**, an X Windows based

bitmap/picture editing tool. Contact Dave via US Mail at P.O. Box 1352; Mountain View, CA 94042 or via email at either koblas@netcom.com or koblas@sgi.com.

buzzerd: Automated Systems Monitoring with Notification in a Network Environment

Darren R. Hardy & Herb M. Morreale – XOR Network Engineering, Inc.

ABSTRACT

As a comprehensive, automated systems monitoring and notification system, *buzzerd*'s goal is to detect system problems and provide accurate diagnostic information for many hosts on a network. This system discovers potential problems as they develop which helps to keep them from growing out-of-control, therefor reducing the chances of system downtime. The *buzzerd* system includes various notification methods, as well as configurable host specific monitoring. A prototype of *buzzerd* has been monitoring the Colorado SuperNet, Inc. network for many months and has proven to be an effective systems administration and evaluation tool.

Introduction

In nearly every technical magazine that covers topics related to UNIX, there is at least one advertisement from some company that promises to provide timely technical service to "minimize costly downtime." It is true that large multi-user machines are hard to keep "up" all of the time, and when they are down, several high-paid employees won't be able to accomplish their work. A system that reduces downtime by catching problems before they become catastrophic would help keep systems running, thus providing a productive environment for all users and reducing the costs of expensive downtime.

The idea of "continuous uptime" and "discovery before failure" is not new to the system administration world. In the past, there were a few basic ways in which downtime was reduced. Probably the most well known method for keeping systems running is the "overworked system administrator." It is not unusual for a systems support person to work 50-60 hours a week. Some companies have been generous enough to staff the support department with enough people so that employees are not overworked. In either case, the basic premise is the same: the more hours spent monitoring and tuning a system, the less downtime expected. This method is expensive, but it works.

Another way to keep systems running is to write simple programs that are run periodically from a schedular such as *cron(8)*, and test certain system conditions. These programs can often take advantage of SNMP queries, and are designed such that a system administrator is notified via logs or electronic mail. This method is generally successful, but it requires that each site write system specific programs and devise some means of timely notification. Furthermore, many times when a problem is discovered, the system support person must be informed immediately, something that log files and electronic mail can not guarantee.

The ideal way to keep systems running would be to install a software system that continuously monitors all important system process, notices problems when they first develop, intelligently designs a solution, notifies a support person about the problem with a recommended solution, then takes corrective measures automatically after approval. Unfortunately, since multiuser environments like UNIX are large and complicated, an expert system such as this is not widely available.

Until intelligent systems are publically available for system administration tasks, the best solution to the downtime problem is a system which catches problems as they develop on critical machines throughout a network, then notifies the appropriate systems support staff immediately. The *buzzerd* network and system monitoring software developed by XOR Network Engineering, Inc. follows this basic approach and has proven to be an important component in providing high quality technical support.

Overview of the *buzzerd* System

The *buzzerd* system performs two primary functions: system monitoring and problem notification. Monitoring is handled by the *sysmond* subsystem. *sysmond* is run on each host throughout the network and periodically executes programs which check specific system characteristics. When *sysmond* detects a problem, it relays the information to the *buzzerd* daemon. This process is responsible for "paging" the system administrator(s) by means of electronic mail, digital or alpha-numeric pager, or simply by logging the information. The notifier also tracks the history of the problem, and will "repage" if the problem appears to be escalating or is not

being acknowledged. Both the notification and monitoring subsystems are modular and extremely configurable.

Monitoring

The *sysmond* system has a standard suite of monitors that check a number of important system processes and characteristics such as system load average, network connections, rapid growth of important log files, electronic mail flow, disk space availability, etc. All of the monitors can be tuned to each host's specific needs. For example, one host may normally run with a load average under three, while another system may be extremely I/O-bound and have load averages closer to ten. The load average monitor checks for "high water marks," which may vary between hosts. When a particular site requires a unique monitor, it can easily be incorporated into the standard collection.

Notification

The notifier can be configured to meet any number of a system administrator's requirements. As previously mentioned, problems can be relayed in a number of different ways, but *buzzerd* can also be setup to use different means of notification depending on the type of problem, the severity of the problem, or the time of day. A typical scenario follows:

> During the day, Beth likes to be notified of all non-critical problems via electronic mail, but after she leaves work at 5:00pm she wants all non-trivial problems to be passed on to her digital pager. After 12:00am Beth only wants to receive pages for situations considered urgent.

One configuration file enables each administrator to customize the notification system for their needs. *buzzerd* also has basic configurable options regarding problem filtering, time-to-wait between "re-pages," and how to handle escalating problems.

Utilities

A few programs in the *buzzerd* system are not related to monitoring or notification, but provide important functionality.

Installation Tools

Maintaining the various *sysmond* daemonsinstalled on a network is a difficult task with many scalability issues. Adding a monitor or changing a parameters might require updates on hundreds of hosts. Therefore, the installation of *sysmond* on numerous hosts as well as the maintenance of the host-specific *sysmond* configuration files is controlled by an interactive program called *ibuzz*. *ibuzz* will install configuration files on each *sysmond* host and notify the running *sysmond* of any changes.

Acknowledging and Viewing Pages

Once a user receives notification from *buzzerd*, *ackpage* can be used by systems administrators to access detailed diagnostic information and data regarding the placement of the problem with *buzzerd*. After the problem is resolved, the user acknowledges the page, removing it from *buzzerd*'s list.

Specifically, *ackpage* supports two commands: *list* and *delete*. *list* shows an abbreviated report for each registered problem and its unique page id. When given a page id as its argument, *list* will display the diagnostic information associated with the problem. *delete* simply removes a problem from *buzzerd* and logs who removed the problem.

Verifying sysmond

checkind resides of the same machine as the notifier and is responsible for making sure all *sysmond* process on the network are running. When *checkind* notices that a hosts monitoring system has not "checked in" recently, it contacts *buzzerd* so a support person can be notified.

Support for Users Contacting Support Staff

Users can page support persons directly through an easy-to-use program, *ezpage*. Users can choose to page a particular person, or page regarding a specific issue. Person initiating a page will be asked to enter a brief description of their problem or question. This allows the person who receives the page to find out what the problem is before they return it.

Design of the *buzzerd* System

The entire software system is called *buzzerd*[1]. The *buzzerd* system is made up of a two primary subsystems that communicate via TCP/IP sockets and a well-defined protocol. Within each subsystem, all major components are separate processes. By having each specialized component perform only specific duties, the entire system is easily configurable.

The remainder of this section is dedicated to the inner-workings of *buzzerd,* the notifier daemon. Details of the *sysmond* subsystem are in the following section.

Figure 1 shows the logical flow of the entire *buzzerd* system, including various utility programs.

The *buzzerd* System

Goals and Requirements

The notification system need not only be responsible for passing on problem reports to the system administrator, but it must also maintain the database of reported incidents, and "re-page" when

[1]The name *buzzerd* came about because the primary means of problem notification was originally envisioned to be via a digital pager. We often like to set our pagers in "vibrate" mode so we don't have to hear the annoying beeping. When the pagers vibrate, they make a buzzing sound. It also seemed an appropriate name since a buzzard, a large bird of prey, is known for waiting about for something to go wrong, then make the best of the situation.

a problem appears to be escalating or completely unacknowledged. It needs to support multiple simultaneous connections from processes throughout the network, while avoiding situations where the system might fail and all problems go unreported. Also, facilities need to be incorporated that eliminate the potential for "page storms" where the support staff is overwhelmed with useless information.

While providing this functionality, a primary goal was that *buzzerd* be easily configurable and flexible since each site has unique notification requirements. One site may have a number of system administrators who share duties and do not wear pagers, while another might have only one support person who heavily relies on their pager for performing their job. Abstracting the means of reporting problems from the actual gathering of information makes configurable notification possible.

Finally, since networks are rarely homogeneous, *buzzerd* must be designed with portability in mind.

Given all of these requirements, it was obvious that *buzzerd* should consist of well-defined modules with specific definitions for intercommunication.

Figure 2 shows the overall internal structure of *buzzerd*.

Interprocess Communication

The monitoring and notification subsystems communicate via TCP/IP stream sockets. The basic protocol for interfacing with *buzzerd* is shown in Table 1.

When a process establishes a connection, *buzzerd* forks a child process to handle all interprocesses communication with the connecting process (i.e., a remote or local *sysmond*). As the child process receives information via the socket, the data is passed on to the parent process through a socketpair. By having *buzzerd* fork a child to handle each connection, multiple connections can be processed from different *sysmond* daemons on the network. Also, since the child only handles the direct protocol handshaking with the connecting processes, the parent is non-blocking

Problem Codes

Since reports of problems must be conveyable via digital pager, a simple system of "problems codes" was designed. A problem code is defined as

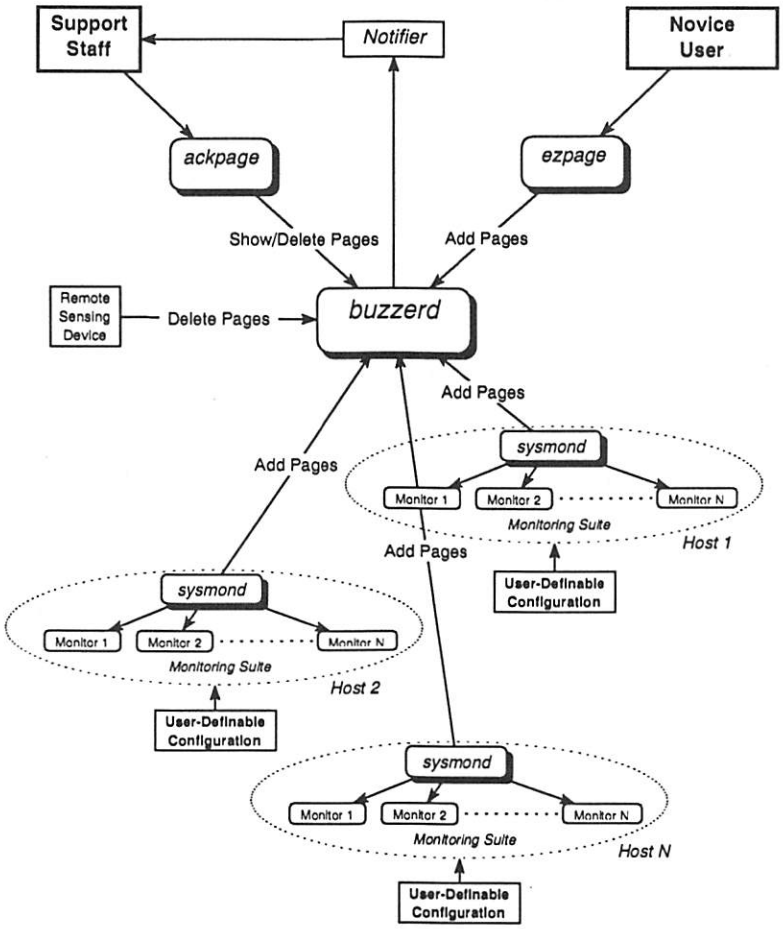


Figure 1: Organization of the *buzzerd* system

a series of three numbers that uniquely identify the host which reported the incident, the monitor that discovered the problem, and the severity of the situation. Each value in the the problem code is separated by a '*'.[2]

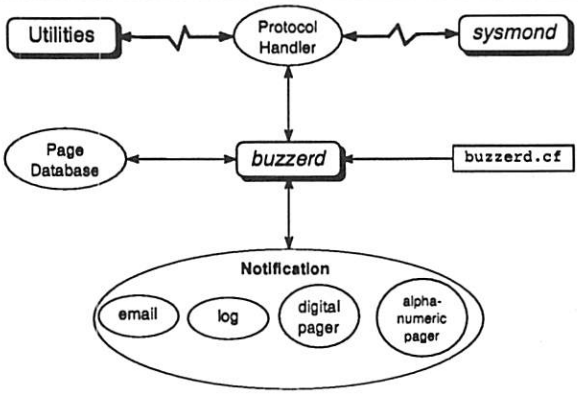


Figure 2: Organization of the *buzzerd* daemon

ADD <problem-code>	Report a problem
DELETE <page> <user>	Resolve a problem (must supply userid)
INFO <page> <data>	Supply supplemental information about a page
LIST	Show all current problems
ONCALL	List person who will be notified of problems
HELP	Show basic help on *buzzerd* protocol
QUIT	Close connection

Table 1: Basic *buzzerd* protocol

For example, when the disk space monitor on the host `teal.csn.org` detects a critical error, it will return the highest severity level 5. If `teal.csn.org`'s assigned number is 1 and the disk space monitor's assigned number is 4, then the problem code would be formed as 1*4*5. Also, the problem codes can be mapped into human-readable text for a concise description of the problem. The human-readable text for the problem code 1*4*5 might be "teal.csn.org: 'Disk Space Availability: Fatal Error, systems are CRITICAL!')." [The *sysmond* section of this paper discusses how the severity levels are decided.]

Furthermore, problem codes provide a unique name space for describing monitored problems and provide a basis for prioritizing problems for notification based on severity levels. This allow for simple searching of the database when checking for duplicate problem reports, as well as notification based on the urgency of the problem.

[2]Asterisks often translate into '-' when sent to a digital pager.

Page Database

When a problem is reported to *buzzerd* with the ADD command, it is stored in a simple database until it is removed with the DELETE command. Each record in the database contains the following information:

- A unique PAGE-ID (used primarily for logging).
- Date and time when the problem was originally reported.
- Problem code reported by *sysmond.*
- Names of people who have been notified of the problem.
- Number of times everyone has been notified.
- Time at which to notify everyone again.
- Additional information regarding the problem.

All information regarding a particular record in the page database is available through the LIST command.

Configurable Notification

When a problem is passed on to *buzzerd* via the ADD command, the process decides for whom the report should be sent, and by which means (electronic mail, digital pager, alpha numeric). The system determines the notification method (i.e., for who and by what method) from a configuration file (*buzzerd.cf*) that is read when the process is initially started. *buzzerd.cf* allows the system administrators to be paged by different methods given the time of day, day of the week, severity of the problem, and various other configuration options. A number of options are available so each support person should be able to come up with a plan with which they are comfortable.

The actual notification is handled by forking a process which performs the required task (electronic mail, page, log). Handling the notification through a separate process keeps *buzzerd* from blocking while waiting for the notification resource (such as a modem).

Filtering

One of most important aspects of creating a notification and monitoring system is the need for some means of avoiding over-notification. For example, one host being monitored might check the reachability of a important network gateway every minute. If that gateway becomes unreachable, *buzzerd* will receive an ADD request every minute. Obviously sending an electronic mail message, or a digital page every minute would be overkill.

Working around redundant pages is actually quite simple since each problem code is unique (host*monitor*severity). When an ADD request is made, *buzzerd* checks the page database for any records with a matching problem code.

The "page storm" that arises when problems begin to escalate is a little more difficult to handle.

Take the simple situation where a system's load average begins to rise steadily over 10 minutes. There are five levels of severity in the *buzzerd* system, so potentially five separate times the system administrator will be notified in a matter of minutes. One way to handle this is to uses "steps" for notification. When using steps, a support person might only be paged at severity levels of 1, 3, and 5 (highest).

Filtering information is a very sticky situation. The system should try to avoid over-notification, however the system administrator should be as informed as possible. This is an area where the best solutions will be discovered only over time.

System Failures

Obviously if the host machine running *buzzerd* crashes, no notification will take place. Furthermore, when a *sysmond* monitoring process fails, all means of reporting problems on that host disappear. The later case is handled by central tracking daemon called *checkind*. When a *sysmond* daemon fails and thus no longer executes a special monitor that "checks in," *checkind* will detect the error and pass an appropriate report to *buzzerd*.

Handling the failure of the notification system is more difficult. Since each *buzzerd* process must have access to a modem for dialing pagers, the number of notification daemons is generally minimized. If *checkind* and checkin monitors are run on multiple notifier hosts (i.e., the machines which run *buzzerd*), the two systems can watch each other. Running multiple *buzzerd* processes is actually somewhat confusing because at this time there is no means to identify which notification daemon "owns" the problem.

buzzerd Implementation

Written in C, *buzzerd* was implemented as a daemon which forks children to handle multiple connections. Each child communicates its information with the *buzzerd* daemon parent process using socketpairs. This approach simplifies sticky data locking issues when children attempt to add their information to the *buzzerd* database. The database of pages and information is kept as an in-memory list with backing store to files in case of catastrophic failure. The configuration file is a regular text file for easy modification and readability.

The *sysmond* System

Design Goals

sysmond, the system monitoring daemon, is a flexible, configurable, detection system for supporting automated monitoring of critical system aspects throughout a network. The sysmond system itself is autonomous, in that it runs entirely on a single host. However, to provide comprehensive network monitoring, many sysmond systems can run simultaneously on different hosts on a network.

sysmond's design is two-fold: a suite of monitoring programs each of which detects and analyzes a different problem, and a daemon which executes designated monitors are regular intervals providing fully-automated monitoring.

The monitoring suite programs detect and analyze various system aspects. Since every computing environment is unique, monitors are flexible and configurable not only giving system administrators greater control over network monitoring, but also allowing them to easily write custom monitoring tools and quickly integrate them into the *buzzerd* system.

The *sysmond* daemon periodically executes its designated monitors, determines whether or not its monitors detected problems, and if appropriate, relays diagnostic information to the centralized *buzzerd* daemon for notification.

Distributed Systems Issues

Since many *sysmond* daemons run simultaneously on many hosts, problems with maintaining the configurations on each host and verifying that all of the *sysmond* daemons are successfully running arise.

Both the *sysmond* daemon and the monitoring suite use a single host-dependent configuration file for various parameters. Using a single configuration file simplifies the maintenance, configuration, and installation of *sysmond* on multiple hosts on a network.

To verify that all of the running *sysmond* systems are working correctly, the *sysmond* daemon periodically executes the checkin monitor which registers with the centralized tracking daemon *checkind*. When a *sysmond* daemon fails and thus no longer executes the checkin monitor, *checkind* passes an appropriate problem report to *buzzerd*.

Host-specific Configuration File, sysmond.cf

The host-specific configuration file which both the monitoring suite and the *sysmond* daemon use for configuration parameters is named *sysmond.cf*. It is a flat file with three fields: the program which uses the parameter, the name of the parameter, and the value(s) for the parameter.

Individual monitors obtain various parameters from *sysmond.cf*. For example, the disk space availability monitor *disk-mon* obtains which disk usage command to use as well as which partitions to monitor and as which levels the partitions are considered too full. The *sysmond* daemon obtains which monitors to execute and at which frequencies from *sysmond.cf*. Global parameters such as the *buzzerd* and *checkind* daemons hosts are also kept in *sysmond.cf*.

Sending the *sysmond* daemon a hangup signal will force it to reread the configuration file and dynamically update which monitors to execute.

Communication with the buzzerd Daemon

Once a monitor has analyzed and detected a problem, the information regarding the situation must be adequately logged with the *buzzerd* daemon so that the appropriate person(s) can be notified. The *sysmond* system supports passing the following information to the *buzzerd* daemon:

- The host on which the problem occurred
- Which monitor detected the problem
- The problem's severity (as estimated by the monitor)
- Diagnostic info gathered by the monitor

Problem codes are used to communicate to *buzzerd* the first three items. Each possible monitored host, each monitor in the monitoring suite, and each predetermined severity level is assigned a unique number. Problem codes are formed by merging the three numbers as follows: numeric code of the host, numeric code of the monitor, and the numeric code of the severity level.

In addition to problem codes, *sysmond* can pass more specific diagnostic information to the *buzzerd* daemon which improves the systems administrator's assessment of a problem. Once the problem is logged with the *buzzerd* daemon using a problem code, the diagnostic information is passed to *buzzerd* which associates the information with the problem code. For example, if disk space monitor finds the `/var/spool/mail` partition is too full and after *sysmond* logs the problem as 1*4*5 (as in the previous example), it would send *sysmond* the following diagnostic information: "/var/spool/mail too full, 90% used, 80% is max, CRITICAL partition." After obtaining this information, *sysmond* then supplies the data to *buzzerd* via the INFO command.

Writing Monitors

Monitors need only to meet a few simple requirements to be successfully integrated with the *sysmond* daemon. In particular, they must return one of the predetermined severity levels as their exit code; they must print any diagnostic information to standard output and optionally via the *syslog(3)* facility; and they must use the *sysmond.cf* file for any configuration parameters. Conforming to these requirements allows the *sysmond* daemon to extract the needed information from the monitors.

Furthermore, the *sysmond* system provides a small library which facilitates compliance with these requirements. Specifically, procedures for logging diagnostic information, calculating severity levels, extracting *sysmond.cf* configuration parameters, and SNMP variable extraction are included in the library.

*Monitor Example: **disk-mon***

The disk space availability monitor *disk-mon* monitors disk partitions for an appropriate amount of free space. In *sysmond.cf*, a high-water mark is associated with each partition. When the occupied space on a partition exceeds the high-water mark, then *disk-mon* will return a severity level that is proportional to the amount that the partition is over the high-water mark. Furthermore, partitions can be marked as critical. When a critical partition exceeds its high-water mark, the highest severity level is returned regardless of how much the partition is over its high-water mark, allowing immediate notification as the *buzzerd* daemon will assign a high priority to the problem.

For example, while running on the Colorado SuperNet host `teal.csn.org`, *disk-mon* detected that the /usr and the /var/spool/mail partition had exceeded their high-water marks as specified in sysmond.cf. /var/spool/mail is marked as a critical partition so the severity level returned was the maximum (5). Figure 3 shows the parameters from *sysmond.cf* and the diagnostic information as logged via the syslog facility.

sysmond Implementation

Written in C, *sysmond* was implemented as a simple, low-overhead daemon which periodically forks children to handle the monitor execution and possible *buzzerd* contact. Figure 4 shows that *sysmond* will fork a *sysmond executor* to handle the entire monitor execution and *buzzerd* contact.

The *sysmond executor* forks a child to execute the monitor program, first setting the process group ID for cleaning up the monitor and any of its children. Once the monitor is running, the *sysmond executor* sets a timeout to prevent monitors from running too long or hanging.

From *sysmond.cf*:

```
disk-mon partition  /usr 85
disk-mon partition  /var/spool/mail 80 CRITICAL
```

From syslog:

```
Aug 24 01:51 teal.csn.org disk-mon[15976]: /usr too full, 92% used, 85% is max.
Aug 24 01:51 teal.csn.org disk-mon[15976]: /var/spool/mail too full, 86% used,
                                           80% is max, CRITICAL partition.
Aug 24 01:51 teal sysmond-executor[15975]: disk-mon EXITED with code 1*4*5
  (teal.csn.org: 'Disk Space Availability: Fatal Error, systems are CRITICAL!')
```

Figure 3: *disk-mon* configuration and output

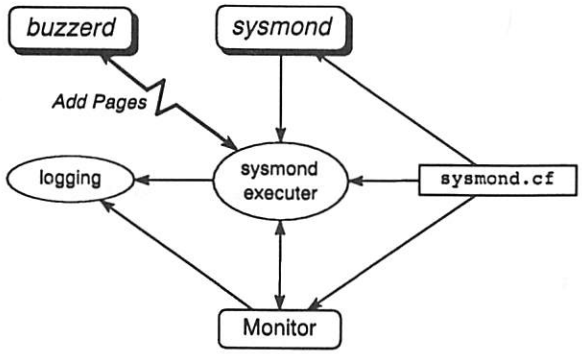


Figure 4: Organization of the *sysmond* system

If the timeout expires before the monitor exits, then the *sysmond executor* kills the monitor and all of its children and then places an appropriate page with *buzzerd*. However, if the monitor successfully exits before the timeout and if the monitor's exit code is non-zero, then the monitor has found a problem and passes a severity level via the exit code to the *sysmond executor*.

The *sysmond executor* then gathers any diagnostic information from the pipe to the monitor and places a problem report with *buzzerd*. Otherwise, the monitor didn't find any problems and the *sysmond executor* exits.

The monitoring suite and the *sysmond* library were written in Perl and C.

Observations on Effectiveness and Utility

The Colorado SuperNet[3] network of 3 Sun SPARCstations running SunOS 4.1.1 provided the initial development and test site for the *buzzerd* system. The Colorado SuperNet hosts are heavily loaded. They are a major electronic mail, USENET, and nameservice hub for the state of Colorado. They also provide support for over 1000 interactive accounts, almost 200 UUCP accounts, and over 100 SLIP accounts.

Developed over 6 months, *buzzerd* has helped keep Colorado SuperNet's network running smoothly. *buzzerd* has been an effective tool for organizing and optimizing Colorado SuperNet's resources by providing timely feedback on problem areas on their systems.

Large Scope of Detection

buzzerd can monitor a large scope of problems since the monitors are generic UNIX programs. Therefore, monitors can use the full functionality of the host to detect and analyze problems.

[3]Colorado SuperNet is a non-profit organization formed by the State of Colorado in 1986 with a mission of promoting use of the Internet for research, education, and economic growth.

Supporting this flexibility has proven to be very useful since it greatly simplifies monitor writing and allows quick integration of local environment. Users need not learn yet another programming language in order to profit from the full functionality of *buzzerd*. They can write the monitors in their favorite language provided that they follow the requirements which insure that it will integrate with *sysmond*.

Keeping Systems Healthy through Monitoring

buzzerd has been effective in determining problems and thus allows the support staff to correct many reoccurrent and elusive problems. With many of these problems resolved, the system and users complain less often, allowing more time for other projects.

At Colorado SuperNet, *buzzerd* has detected subtle configuration problems with our terminal servers, disk space allocation, reoccurring connectivity problems, skewed load distributions, and many other problems. *buzzerd* has been invaluable in quickly detecting and analyzing these problems, allowing corrective measures to be taken. Furthermore, since 24 hour support is required of Colorado SuperNet's computing environment, the digital paging capabilities of *buzzerd* have kept the support staff alert and responsive.

System Load and Vicious Cycles

Although monitoring provides very useful information, the monitoring software should not require many resources to run, but rather it should be optimized to allow the system to accomplish its designated tasks. When a system is overloaded, continued monitoring of the system might cause further problems rather than produce helpful information. Therefore, to help prevent this viscous cycle, *sysmond* takes a couple simple precautions which have proven effective.

First, *sysmond* attempts to ease system load by balancing the execution times for the monitors. For example, if *sysmond* has 5 monitors that execute every 10 minutes, rather than forking 5 children at once, *sysmond* spaces the forking of children to distribute the system load over a longer time period. Therefore, rather than initially running all of the monitors and then waiting 10 minutes to run the next batch, the load balancing only runs one monitor in the first minute, then another monitor in the next minute, and so on.

This simple scheme yields a large reduction in load. Before implementing load balancing, with 25 monitors running at various intervals, the maximum number of monitors that would be run within a single minute was 23. However, with the load balancing, this maximum was reduced to eight.

Finally, *sysmond* monitors *syslogd*. If *syslogd* crashes, *sysmond* will switch to logging to a file rather than via the syslog facilities. In the past, we've experienced serious performance problems

when *syslogd* has crashed and programs try continually to use the syslog facility.

Future Directions

Further Testing

Although *buzzerd* has been monitoring the Colorado SuperNet network for several months, *buzzerd* still should undergo a beta testing phase in which several, larger, heavily-used networks are monitored. Feedback on the systems usefulness and design would be helpful.

Furthermore, the scalability of *buzzerd* needs further testing. Although *buzzerd*'s design supports large networks, real experience can prove otherwise.

Graphical User Interface

Since the monitoring suite extracts so much information from the systems on a network, users could use a graphical user interface which allowed easy viewing of monitoring information. Information such as disk space availability, system load, status of terminal servers, mail and news traffic flow could all be graphically displayed allowing a user to easily obtain a "feel" for the network's state. Also, a graphical user interface for *ackpage* would also help users track current problems.

Remote Sensing Device

Pages can be acknowledged by dialing a remote sensing device that is connected to a serial port. This device enables users to acknowledge that the page has been handled without having to login and run *ackpage*, thus stopping *buzzerd* from re-paging. Similarly, users can use the device to tell *buzzerd* to defer paging until the problem can be resolved. Furthermore, a dialup device such as this could be used accomplish other system tasks such as reboots, power-cycles, etc.

Acknowledgments

We wish to thank David C. Menges for motivating this project and allowing us to use the Colorado SuperNet network as a development and test site. We would also like to thank Trent R. Hein who has provided lots of helpful input into the original design and assistance with many complicated implementation issues.

References

Comer, Douglas E., *Internetworking with TCP/IP*, 2nd Edition, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1991.

Davin, James R., *The SNMP Development Kit: Release Notes*, Massachusetts Institute of Technology, 1990.

Kernighan, Brian W. and Ritchie, Dennis M., *The C Programming Language*, 2nd Edition, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1988.

Leffler et al., *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley, Reading, MA, 1989.

Menges, David C., *Colorado SuperNet Dialin Information*, June 9, 1992, available via anonymous ftp on csn.org:/CSN/reports/DialinInfo.txt.

Nemeth et al., *UNIX System Administration Handbook*, Prentice Hall, Inc., Englewood Cliffs, NJ, 1989.

Stevens, W. Richard, *UNIX Network Programming*, Prentice Hall, Englewood Cliffs, NJ, 1990.

SunNet Manager 1.2 Product Brief, Sun Microsystems, Inc., Mountain View, CA, 1991.

UNIX Programmer's Supplementary Documents, Volume 1, Computer Systems Research Group, University of California, Berkeley, CA, 1986.

Wall, Larry and Schwartz, Randal L., *Programming perl*, O'Reilly and Associates, Inc., Sebastopol, CA, 1990.

Author Information

Darren R. Hardy earned a B.S. in Computer Science from the University of Colorado at Boulder in 1991 and is currently working on an M.S.C.S. specializing in network resource discovery and distributed systems. As a systems engineer for XOR Network Engineering, Inc., he develops network support software and designs, installs, and supports small- to medium-sized networks. Reach him via US Mail at XOR Network Engineering, Inc., 1495 Canyon Blvd, Suite 35, Boulder, CO 80302 or electronically at hardy@xor.com.

Herb M. Morreale has many years experience writing user-friendly system management software. He is co-owner of XOR Network Engineering, Inc., a Boulder-based UNIX/Network consulting company that specializes in helping businesses get started with UNIX, networking, and the Internet. In his current position, Herb is responsible for 24 hour monitoring of the largest dialin UUCP and USENET hub in the state of Colorado, as well as other systems throughout the country. Herb has a B.S. in Computer Science from the University of Colorado at Boulder. Reach him via US Mail at XOR Network Engineering, Inc., 1495 Canyon Blvd, Suite 35, Boulder, CO 80302, or via electronic mail at herb@xor.com.

bbn-public – Contributions from the User Community

Peg Schafer – BBN

ABSTRACT

bbn-public is the repository of software and information of interest to the BBN UNIX computing environment, which is supported by the BBN computing community. With respect to UNIX system administration, BBN is a collection of semi-autonomous divisions that may or may not have organized system administration policies or system administration personnel. Situations range from the lone researcher in a lab with a system straight out of the box, to groups that maintain well-managed systems. In the past, sharing of information or binaries seldom occurred between groups. However, individuals in the user community were producing valuable services which could benefit the whole BBN community. The BBN-wide UNIX support group covers the global services such as mail and printing, but is not chartered to provide development of items that benefit specific portions of the community. The problem: How to get the community to share information, increase communication, benefit from the work of individuals, and reduce replication of work? All this, of course, without spending money or time! bbn-public is our answer.

The Problem

UNIX system administration across Bolt Beranek and Newman (BBN) is fragmented by political, functional, accounting and geographical boundaries. The company is divided into semi-autonomous divisions, each with its own goals, uses for computers, and most importantly, funding. The corporation BBN Inc., is the "glue" which holds these divisions together. Within BBN Inc., the Distributed Systems & Services group (DSS) offers UNIX support services on a contractual basis to groups within BBN. In addition, the DSS provides several services which benefit the BBN computing environment as a whole (e.g., e-mail, printing, networking, etc.).

At every site there are individuals who produce services or tools which could be of use to the whole community. I'm happy to say that BBN has a wealth of such individuals. However, there was little communication between groups, which resulted in much duplication of work. For example, each group was compiling their own versions of emacs.

The problem: How to get the community to increase communication, benefit from the work of individuals, share information, and reduce replication of work – both economically and efficiently?

Our Solution

We created the /bbn-public file system. Advertised to the BBN computing community as a software library and software kiosk, the DSS invited members of the BBN computing community to contribute. The definition and polices of bbn-public were purposely simple and brief. We wanted a simple design with few restrictions in order to facilitate dynamic utilization and growth. Contributors could be any member of the BBN Computing Community – system administrators or random users.

The key element is that contributors must agree to support their contribution. This enables users to contribute to the quality of the local computing environment, and allows the computing environment to be enhanced in ways the system administration staff may be unable to provide.

How bbn-public is Utilized and Growing

At this moment there are more than 17 software packages available on bbn-public. These include such helpful bits as bash, ntpdate, perl, TEX, khoros, and so on. In addition, the Prime Time Freeware CD, which contains a wealth of freeware and documentation, is mounted on /bbn-public.

Because bbn-public is not just another bin directory, it has grown to meet the dynamically growing needs of the BBN computing environment.

- After a security alert, we created a directory for security-related items, in which are placed the CERT announcements, shell scripts, a COPS directory, checksums for various binaries for several architectures and operating systems, an RFC or two on security topics, and security patches.
- The "doc" directory holds BBN specific polices, vendor white papers, and information on various presentations. (This area will receive a large amount of attention in the future. A guide to the BBN computing environment info server will be placed here.)
- bbn-public includes a statistical profile of the BBN computing environment, useful for

inclusion in reports, proposals, etc.

- The DSS announces all policies, alerts, new services, etc., on their bboard. The postings are automatically archived to a mail box on bbn-public. This is a useful historical record.
- There is a list of helpful resources and phone numbers around BBN.
- System administrator tools are very popular on bbn-public:
  - Sun's Pipeline tools have been installed to aid in the transition to Sun OS 5.0.
  - All of the X11R5 and Xcontrib sources can be found on bbn-public.
  - BBN's patch database for the millions of patches from all vendors has also been quite helpful.
- BBN has discovered bbn-public to be an internal distribution point for software developed in house. Beta releases of software can be placed on bbn-public for perusal by the user community.

Not long after bbn-public was started, contributors asked what path should be used when compiling binaries for general use throughout the BBN computing environment. Since we did not have any path standards in place at BBN, a working group was formed in conjunction with DSS and the user community to propose paths for the general computing environment. This was a very successful venture; path standards can now be found as /bbn-public/doc/standard-paths.

Nity Gritty Implementation Details

How does an item get nominated and installed? Very simply, a potential contributor contacts the DSS. The software must be accompanied by a README which states the contributor's name, and the level of service the contributor wishes to provide. After reviewing the software, DSS installs it and the README on bbn-public, and then updates the CONTRIBUTORS file and announces its availability via the bboard.

Members of the BBN computing community may access the bbn-public file system via NFS mounts or a visitor login. bbn-public was not designed as a bin directory for export to the BBN computing environment, as the DSS does not own or maintain a corporate file server. bbn-public files and directories are purposely laid out to inhibit execution. The host machine is a Sun sparc II, which can easily handle the network traffic for bbn-public, but it is not the appropriate hardware host for a corporate-wide file server. The bbn-public file system is simply a 1.3 gig SCSI disk drive.

For security, bbn-public is exported read only to the net group "all-bbn" which includes all the UNIX-based machines at BBN. Every night a shell script takes an updated list of systems, and removes the names of selected machines. A list parser then separates the list into groups of suitable size for NIS maps, generates a new /etc/netgroups file and remakes the netgroups NIS map. Hence, old systems are dropped automatically from the list and new ones are added without human intervention.

Why go to all this work? Why not just have a simple anonymous ftp directory? Initially, we installed anonymous ftp, but found we could not restrict outside access. If we kept anonymous ftp, we would be unable to install the BBN developed software. We desired an interactive interface for users to peruse the software before actually buying (installing) it on their systems. The visitor login enables a user from within a restricted shell to evaluate the software or review the item of interest. The visitor may then ftp the software back to their machine.

As bbn-public has been in place for only five months, we have not had any problems with the file system filling up or contributors abandoning their contributions. When the disk fills up the computing community will evaluate the packages' worth and delete items deemed expendable. If the contributor no longer wishes to support a package, he or she is requested to find someone else who will maintain the package. If no replacement can be found, the software will be evaluated for its usefulness to the BBN community.

Future Directions

In a word – more. We plan to add more software as the community generates it. Databases of several types have been proposed in addition to dictionaries, thesauri, and perhaps encyclopedic material. The documentation directory will receive attention within the next six months. A guide to computing at BBN has been proposed and will reside there. A hoped-for CD juke box will hold the growing number of freeware CDs and RFC CDs. Info servers to service requests for information from these databases will be included. Got an idea? I'd love to hear it! peg@bbn.com

Conclusion

bbn-public is a roaring success. Here, success is measured in two ways: 1) The file system is mounted by approximately 145 systems at any one time. bbn-public is a low cost, low time expenditure effort which turned out to fulfill a need in the computing environment. Due to its flexibility, it will go on to fulfill these developing needs.

2) For me, and most importantly, bbn-public serves as a focal point by which all users may contribute to the enhancement of the BBN computing environment. For too long now, system administrators felt they had to "do it all". But there are gold mines of resources within our user community, and bbn-public allows the DSS to tap it. Moreover, by

their contributions, members of the computing community feel involved in influencing the direction of computer support at BBN.

Acknowledgments

This work is a co-operative effort. Many thanks go to: Mike Accetta (from CMU) who contributed the concept of system administrators drawing from the talents of the user community, Margaret Metcalf who first suggested a large scratch space anyone could "dump stuff on", Frank Corcoran, the manager of DSS, who fought and won the budget battles for us, the members of DSS; Pat Harmon, Betty O'Neil, Pei Ren, Pam Andrews, Ed Eng, Frank Lonigro, John Orethoefer and David Nye, the many contributors of bbn-public, who cared enough to share their good efforts and most of all, the BBN computing community for making it a success.

Prime Time Freeware can be reached at 415-112 N. Mary Ave., Suite 50, Sunnyvale, CA 94086 USA, Tel: +1 408-738-4832, ptf@cfcl.com.

Author Information

Peg Schafer's title is Senior Systems Programmer for the Distributed Systems and Services group at Bolt Beranek and Newman Inc. Snail mail: 10 Moulton Street, Cambridge MA 02138, 617-873-2626 peg@bbn.com

user-setup: A System for Custom Configuration of User Environments, or Helping Users Help Themselves

Richard Elling & Matthew Long – Auburn University

ABSTRACT

Large sites often have the problem of too much software, too many users, and not enough support staff. This paper describes user-setup: an easy to use system which allows users to customize their environment by selecting their preferred applications. The system is easy to administer, scales well, and does not limit advanced users. user-setup generates correct by construction C-shell "dot" files and customized OpenLook Window Manager menus. Thus, a user can have a personally customized environment that works without having to learn shell languages, X window manager menu languages, or text editors. The infrastructure is based upon the Modules package [Furlani91].

Introduction

Typical UNIX shells offer an often bewildering assortment of options and customizations. The shell is the most intimate interface between the system and its users. The shell is also one of the most complicated utilities to configure. Any mistake in the shell configuration files could render the shell completely useless. For the naive user, this is often catastrophic and contributes to the myth that UNIX is difficult to use. From the administrator's point of view, deciding on the perfect shell configuration becomes increasingly difficult as applications are added, upgraded, or phased out. For large sites the concept of editing every user's shell configuration file is ludicrous.

user-setup attempts to solve the problem by allowing the users to select applications for themselves. user-setup uses a simple menu interface with on-line help to guide the user through a list of available software packages. The user can review the packages and select those they wish to try. user-setup generates correct by construction shell configuration files that give users access to the applications they desire. Most importantly, users can run user-setup to safely change their environment at any time without having to consult the computer center support staff.

The Problem

Our problem is really one of resources: how to provide access to the large diversity of applications for a large number of users in a ubiquitous environment with a limited support staff.

Too Much Software

There is a **lot** of software in the universe. It is not unusual for a single site to have several word processing, publishing, math, or spreadsheet packages. In the engineering world, there are dozens of drafting, modeling, and analysis packages. Worse yet, many sites support several different window systems. Some of the packages only work in some of the window systems or perhaps only certain versions of the package under certain window systems. In many cases a new version of the application arrives which cannot immediately replace the existing version due to window system changes, operating system changes, or new features. So there may be several different versions of the same application which require support.

New Users

New users are perhaps the one true test of any system. New users often don't know what a shell is, what an editor is, what a "dot" file is, what a path is, or why they should bother with them. Often users are told to get an account and start using it. A typical new user will not know what applications are available or how to gain access to them. A simple typographical mistake in a shell configuration file could render the user's session unusable. Learning how to recover from a broken shell configuration should not be the second lesson taught to a new user. Frantic calls to the help desk are assured if new users are required to edit shell configuration files. user-setup solves this problem by creating correct shell configuration files for the user.

Guidance Conflicts

Guidance conflicts arise when several people give conflicting instructions to a naive user. For example, one instructor might require the users to run an initialization script to setup their environment. This script will likely conflict with the script that another instructor requires of the same users. All users are strongly encouraged to use user-setup and it has become the only help desk supported method of changing their environment.

Dot File Virus

There is a very nasty virus that can spread among users like wildfire: the "dot file virus." This virus is easily spread from user to user by such words as, "to run application ABC, just copy my dot file." Once a broken dot file virus starts, it can be very difficult to stop. Meanwhile, dealing with users bit by it can consume vast quantities of help desk time. In such cases, the user can always run user-setup to recreate a stable environment. As long as the modulefiles are properly maintained and updated, "dot" file viruses can be virtually eliminated. Now users say, "to run application ABC, just run user-setup."

Requirements

The design goal of user-setup is simple: provide a tool for users which will correctly configure their environment for whatever application they want to use. To reach this goal, several requirements were identified:

User-friendly

First and foremost, user-setup must be user-friendly. Not surprisingly, user-setup is one of the first applications run by new users. Since first impressions are important, we tried to make user-setup as friendly as possible. The idea was to make it so easy to use that users would be willing to run it again at any time to change their environment.

Idiot Proof

Obviously, user-setup must be idiot proof. No user-setup actions can affect the user's current environment. No "dot" files are modified until the user applies the environment changes. All new "dot" files are created in a temporary directory and are not copied into the user's home directory until they are completed constructed. All existing "dot" files are backed up before the new ones are copied. user-setup must not rely on the current user environment setup to run. Thus, if the user's current environment is totally trashed, user-setup must be able to recreate a stable, working environment.

Modules Infrastructure

user-setup uses the Modules package as its infrastructure. A modulefile is created for each application which contains the configuration information needed to run the application. The configuration information is typically comprised of changing the PATH, MANPATH, and other environment variables. The modulefiles are written in Tcl [Ousterhout90] which allows sophisticated conditionals and control over the environment changes. An added benefit of using the Modules package is that changes can be made to modulefiles which reflect changes in applications or application availability without forcing users to run user-setup again.

Low Maintenance

user-setup must be easy to maintain. The initial configuration of user-setup is relatively simple. The major portion of the maintenance involves the modulefiles. At least one modulefile must be created for each application. By convention, we include a separate file which contains the application description. This file can be displayed from the modulefile. For applications which can be started via olwm menus, two additional files are required which contain the menu information. All of these files are plain ASCII text with revision control by SCCS. Once the modulefile has been added to the modules directory structure it is ready to be accessed by user-setup.

Application Transition

A mechanism must exist to safely transition to new versions of applications as they become available and as the old versions are removed. There are several strategies that can be used by cleverly writing modulefiles. The modules directory structure is constructed so that each application is represented by a directory which has a modulefile for each version. For example, the X11 application directory may have modulefiles for R4 and R5. When R4 is deleted, the R4 modulefile could be rewritten to notify users of the change, run R5 instead, or even run user-setup.

Power Users

user-setup must support power users. Power users are traditionally very leery of anything which will touch their "dot" files. Power users also tend to make rather substantial changes to their "dot" files which they'd like to keep. user-setup provides a personal customization area in which a user may place shell commands. The personal customization area will be copied verbatim into the new .cshrc file. The personal customization area is located near the end of the .cshrc file so that any commands placed there will override the actions of any of the modules. The approach we take is that user-setup should be so good that power users will want to use it.

Lost Server Survival

A reasonable fail-safe for when modulefile servers are down must be included so that a user will not be thrown into a completely useless state.

Heterogeneous OS Support

user-setup must enable us to provide a consistent environment across major OS releases and hardware platforms. Applications which are not available for the current platform must gracefully inform the user without destroying the user's session.

Implementation

user-setup(1) is implemented as csh script. It provides an easy to use, character based, menu driven interface to the Modules package. Some

modifications were made to the Modules package version 1.0 in order to support additional file types in the Modules directory structure. Once the user's environment has been created using user-setup, the user may make additional changes using module commands. Most users choose to use user-setup rather than module commands.

Configuration Files

user-setup was designed to be easily maintainable. Besides the modulefiles, only five configuration files are needed: a default .login file, prologue.cshrc, default customization, epilogue.cshrc, and a question order file.

Default .login File

The default .login file is copied into the user's home directory unaltered. The .login file is primarily used to start the appropriate window system. An environment variable is passed to the .login file to provide the command for starting the desired windowing system. A windowing system will only be started if the user is logged into the console. It is not anticipated that a user will modify the .login file so no provision for keeping changes is provided. The user's existing .login file will be moved to .login.bak.

.cshrc File Prologue

The prologue.cshrc file contains: any environment settings (e.g., umask, limit, other shell variables), the Modules package initialization, and the loaded modulefiles which are edited by user-setup. The prologue includes a fail-safe mechanism so that if no modulefiles are found, the user is not left in a helpless state.

Personal Customization Area

A default personal customization file is provided for initial account creation or first time users. If the user's current .cshrc file does not contain a personal customization area, then the default will be inserted. If a user has made a mistake in the personal customization area, the -nc option user-setup will force the default to be used.

The test for interactive shell is made in the personal customization area. This allows users to add shell commands for all shells as well as for interactive shells.

.cshrc File Epilogue

The epilogue.cshrc file contains any commands which will be executed after the personal customization. We place a network message of the day reader here.

Question Order File

The us2.order file is used to define a required order of initial selections. The first required selection chooses which flavor of UNIX is desired: BSD or System-V. The second required selection is the windowing system. This ordering assures that the other applications will have a known base to work from. This also simplifies the modulefile prerequisite list since each windowing system requires a UNIX base. Thus, a modulefile typically needs only to require the necessary windowing system to work properly.

Modules Package

To help ensure fail-safe operation when application servers are down, the Modules package is loaded in the /usr/modules directory on each workstation. The Modules package constructs a MODULEPATH which contains the names of directories which contain modulefiles. To ease maintenance and help ensure coherency, at least two directories will be specified in the MODULEPATH: /usr/modules/modulefiles and /usr/local/modules/*OS/-app-arch/release*. The /usr/modules/modulefiles directory contains the modules required to gain access to the UNIX commands resident on the local disk (e.g., /usr/bin.) The /usr/local/modules/*OS/app-arch/release* directory contains modulefiles which are applicable to the OS, application architecture, and OS release. The /usr/local directory is automounted from several servers and contains the majority of the modulefiles. If no /usr/local server is operational, the /usr/modules modulefiles will be sufficient to allow the user to login in a somewhat working state. Additional MODULEPATH directories may be added by a user, instructor, or workgroup and will be automatically accessible to user-setup.

A few minor modifications to the original Modules package were made. These changes dealt with the view of available modules. A program was written to recursively search a directory and list any files which had the Modules magic cookie. This addition allows arbitrary directory depth as well as the ability to place other files in the same directory structure. This also allowed us to consolidate all of the files relating to an application (e.g., olwm menu generation information) in one place.

The description of each application is kept in a file separate from the modulefile. By convention, the module display command will also display the contents of the description file. The convention allows a simple shell script to be written which generates a list of the descriptions of all available applications. This will also come in handy for future versions of user-setup.

Related Programs

There are several programs which can be used to modify a user's environment which was created by user-setup. *make-olwm-menu* (1) will generate a custom applications menu for the user based upon which modulefiles are selected. *default-printer* (1) will safely add or change the PRINTER environment variable. *prmap* (1) is an X-Window front end to *default-printer* (1) and also provides information on the location and type of network printers available.

Future Work

X-Window Interface

Development is underway on an X-window based version of user-setup. It is anticipated that the X version will be more popular than the terminal based version. The goal of the X version will be to provide a drag and droppable icons for each modulefile. A user will be able to peruse a collection of modulefile icons and select those that are desired.

Performance Enhancements

As the number of applications grow, the modulefile directory structure also grows. This affects the user-setup startup time since user-setup searches all of the directories in the MODULEPATH for modulefiles. A modulefile cache is being considered.

As part of the X-Window interface, a C++ class library for accessing modulefile description and dependency information is being developed. This development may lead to a C++ terminal version of user-setup which will replace the csh script.

More Shells

user-setup currently supports csh. tcsh is easily supported via user-setup since it is backwards compatible with csh. Bourne and Korn shell support will be provided in a future version.

Example Session

In the extended example shown below, the user would like to add a statistics package to their environment.

```
guest% user-setup

Welcome to user-setup!

user-setup is a menu driven program that helps you to create and modify
your environment.  You can run user-setup at any time to make changes in
your environment by adding or removing applications.  You may quit at any
time and the no changes will be made to your account.

To use user-setup, select the applications you wish to add or remove from
your account.  You will be shown a brief description of the application
and asked whether you wish to use it.  When you are finished selecting
applications, select "apply" from the main menu.

Searching for available Modules.....done.

Press [return] to continue.

Application Classes

          1 - apply
          2 - help
          3 - show
          4 - demo
          5 - math
          6 - misc
          7 - office
          8 - unix
          9 - win

Please select a Class of Applications [?,??,q]: math

Applications for math

          1 - done
          2 - mathematica
          3 - matlab
          4 - sas
          5 - splus
          6 - tksolver

Please select an Application [?,??,q]: 4

------- /usr/local/modules/sparc/SunOS/4.1.1/math/sas/6.07 -------

                        SAS version 6.07

The SAS system is an integrated system of software providing complete
control over data management, analysis, and presentation.  There are
```

```
two components of the SAS system available on the College of Engineering
Sun network: SAS/GRAPH and SAS/STAT.

Prequisites are (ORed):  win/openwin/3.0 win/openwin/2.0 win/X11/R4 win/X11/R5
Append /vol/sas to PATH
Append /vol/sas/utilities/man to MANPATH
-------

Do you want to use sas/6.07 [y,n,q] y

Applications for math

          1 - done
          2 - mathematica
          3 - matlab
          4 - sas
          5 - splus
          6 - tksolver

Please select an Application [?,??,q]: done

Application Classes

          1 - apply
          2 - help
          3 - show
          4 - demo
          5 - math
          6 - misc
          7 - office
          8 - unix
          9 - win

Please select a Class of Applications [?,??,q]: 1

Applying environment changes...

math/sas/6.07
unix/sysv
win/openwin/3.0

Do you wish to apply these changes ? [y,n,q] y

Finished making changes to your environment.  You may need to logout and
log back in for the changes to take affect.

You may also wish to create a custom menu based on your environment.  To
do so, run the command "make-olwm-menu" or select the "Make new menu"
entry from the "User services" menu in OpenWindows.

Press [return] to continue.
Cleaning up temporary files.
```

Now the user's environment is properly configured to run SAS. Now the user creates a custom menu for the new environment:

```
guest% make-olwm-menu

Hello.  I am ready to customize your OpenLook menu based upon the
modules you have loaded.  The menu will be custom tailored to the
applications you've selected.  We hope this will help make the
system easier to use.

Feel free to run this program any time to update your menu.

Do you want to update your menu now [y,n,q] y

Creating new menu...
```

```
No menu entry found for application: unix/sysv
Adding menu for application: math/sas/6.07
Adding default window system menu win/openwin/3.0.menu...
Renaming your existing .openwin-menu to /home/guest/.openwin-menu.bak
make-olwm-menu: removing temporary file.
```

The user's environment is now completely configured with a custom menu for running SAS.

Observations

user-setup version 1 was installed in September 1991. Version 1 was not menu based nor based upon the Modules package and resulted in a system which worked, but was not pleasurable to use nor maintain. user-setup version 2 has been in use since May 1992. Version 2 is menu driven, easy to use and maintain, and based on the Modules package. Approximately 2,500 users have used user-setup to gain access to more than 100 major applications in a ubiquitous environment. During this time we have observed a number of interesting phenomena.

Help Desk

First and foremost, users stop bugging the help desk asking how to get to applications. The standard help desk response is "run user-setup." This has significantly reduced the workload on the help desk.

Guidance

Instructors and workgroup leaders no longer need to provide elaborate instructions on how to modify user environments to gain access to required applications. Their standard response is also "run user-setup." This has proven to be an enormous benefit since it is typical that a user may be taking several classes and often the instructors would not take this into account with their notorious instructions.

Explorers

Users are able to easily discover and use available software. This has helped reduce the number of requests from the user community for new software which is functionally equivalent to existing software. This has also reduced the tendency for users to reinvent or download software which is already installed.

Users can try out new software. We are constantly installing new software and demonstration software. With a few edits we can create new modulefiles which will automatically be available via user-setup. Often creating the new modulefiles is easier than installing the software.

Training Gurus?

Many users don't know UNIX commands, what a "dot" file is, or even how to start applications from the command line. All they need to know is how to login, run user-setup, and run their applications from the custom menu. We believe this is a good thing.

OLWM Menu Madness

Some users are disappointed when their application can't be started from the custom OLWM menu. Unfortunately, some applications are just impossible or impractical to start from menus. For these, we don't provide any custom menu entries. Perhaps in the future we could pop up a brief "how to get started with the application" help screen.

New User Dilemma

There is a dilemma facing administrators when creating new accounts: should the user be forced to run user-setup, or should it be considered an option? If it is forced upon new users, they often are scared and confused and may easily give up. If it is always optional, what initial collection of modules is needed? Specifically, should a windowing system be part of the initial configuration? If so, which one?

Conclusion

user-setup enables users to change their environment easily. Applications can be added or deleted from the user's environment without the user having to learn editors or shell languages. New applications can be installed and made accessible to the users via user-setup by creating a simple modulefile for the application. This helps solve the problem of having too much software, too many users, and not enough support staff.

Availability

user-setup and its associated programs are available for anonymous ftp from ftp.eng.auburn.edu [131.204.10.91]. If you don't have internet access, contact the authors to arrange other media. All code is freely distributable.

Acknowledgments

We would especially like to thank John Furlani who wrote the Modules package. John's timely LISA-V paper allowed us to stop our reinvention of Modules early in the design cycle.

We would also like to thank the faculty, staff, and students of Auburn University who used user-setup version 1 and provided input to the Modules based version 2.

Bibliography

[Furlani91] Furlani, John L., *Modules: Providing a Flexible User Environment*, 1991 USENIX Large Installation System Administration V Conference Proceedings.

[Ousterhout90] Ousterhout, John K., *Tcl: An Embeddable Command Language*, 1990 Winter USENIX Conference Proceedings.

Author Information

Richard Elling is the Manager of Network Support for the College of Engineering at Auburn University. He graduated from Mississippi State University with a BSEE in 1986. He has 11 years of experience in writing and maintaining computer aided engineering tools. His current mission is to develop an awesome engineering environment at Auburn. Reach him via U.S. Mail at Auburn University, Engineering Network Services, 103 L-Building, Auburn University, AL 36849. Reach him via telephone at (205)844-2280. Electronic mail sent to Richard.Elling@eng.auburn.edu is preferred.

Matthew Long is graduating senior at Auburn University. He is available for employment 1/1/93. Reach him via U.S. Mail at Matthew Long, 110 Cedar Crest Circle, Auburn, AL 36830. Reach him via email at Matthew.Long@eng.auburn.edu.

NAME

user-setup – interactively configure a user's environment

SYNOPSIS

user-setup [*–nc*]

DESCRIPTION

user-setup is an interactive program designed to help users configure their environment. **user-setup** will generate a new *.cshrc* and *.login* file for the user. The old *.cshrc* and *.login* files will be saved as *.cshrc.bak* and *.login.bak* respectively. The user can quit at any time without the changes taking place. The *.cshrc* and *.login* files are guaranteed to be correct by construction.

user-setup works as a menu driven front end to the **Modules** package. The **Modules** package allows for the dynamic modification of a user's environment via *modulefiles.* Each major application will have a corresponding *modulefile* which will contain the configuration commands for adding the application to the user's environment. **user-setup** is used to select the initial collection of *modulefiles* to be used upon logging in or starting a shell. This same list of *modulefiles* can be used by **make-olwm-menu** to generate a customized OpenLook Window Manager, **olwm,** menu which contains easy to use menu entries for the *modulefiles* selected.

user-setup will present the user with a top level menu. Under this menu *modulefiles* are grouped using a *class/application/version* designation. The top level menu also contains an *apply* entry to apply the changes to the user's environment, a *help* entry for online help, and a *show* entry to show all currently selected *modulefiles.* Menu entries can be selected by number or a unique string of characters matching the menu item name. For example:

1 - apply
2 - help
3 - show
4 - math
5 - office
6 - unix
7 - win

Each *modulefile* may require the previous selection of another *modulefile.* For instance, a windowing application will require the prior selection of the appropriate windowing system. To assure an orderly initialization **user-setup** requires an entry from the *unix* class first followed by an entry *win* class. This forces a user to select the flavor of UNIX, BSD or System-V, and a windowing system before any other *modulefiles.* Other *modulefile* prerequisites are handled as each *modulefile* is selected.

user-setup checks for *modulefile* conflicts when a *modulefile* is selected. For example, if the *modulefile win/X11/R4* is already selected and the user tries to select the conflicting *modulefile win/X11/R5* **user-setup** will recognize the conflict, notify the user, and disallow the *win/X11/R5* selection. It is the responsibility of the user to correct any conflicts.

A typical **user-setup** session involves selecting a collection of *modulefiles* then applying the changes to the user's environment. When a user selects a *modulefile* they will be shown a brief description of the application and asked whether to add the *module* to their environment or not. This is a simple yes or no question. If the answer is yes, then the *module* will be checked for conflicts and added to the list of *modulefiles* to be applied. If the answer is no, then the *modulefile* will not be added to the list. If a *module* was previously selected, then a no response will remove the *module* from the list. The list can be previewed with the *show* command in the top level menu. Once the user has selected the desired *modules* then the changes can be applied with the *apply* command.

Applying the changes will cause **user-setup** to completely regenerate new *.cshrc* and *.login* files. The *.cshrc* file will contain a personal customizable section. This section will be copied verbatim into the new *.cshrc* file's user customizable section. This allows users to make adjustments to their *.cshrc* file which will not be lost the next time they run **user-setup.** The user customizable section is clearly marked and located near the end of the *.cshrc* file so that the user can override any variables set by **user-setup.** The *–nc* option will cause the system default personal customization to be included rather than the user's. This option is useful when there is a mistake in the personal customization rendering the environment useless.

user-setup can be run again at any time to add new applications to the user's environment or to change windowing systems.

Unlike the **module** command, **user-setup** only modifies the user's "dot" files.

The user will typically need to logout and then login for the changes to take place.

FILES

~/.cshrc	contains most environment configuration.
~/.login	starts the desired window system on login.
/vol/info/user-setup2	directory which contains the **user-setup** files.

SEE ALSO

csh(1), module(1), olwm(1), make-olwm-menu(1)

AUTHORS

Richard Elling <relling@eng.auburn.edu>,
Matthew Long <long1@eng.auburn.edu>

BUGS

The *.login* file is completely regenerated from scratch. Any user customizations to the *.login* file will be lost.

NOTES

Users tend to ignore the descriptions of the *modules;* causing them to miss important information.

Currently only the C-shell, **csh,** is supported.

There is a **lot** of software available on the College of Engineering network. Nobody knows how it all works. Each application has its own demands and quirks. Every effort will be made to keep the *modulefiles* and their actions current. Feedback is very important in maintaining an accurate set of questions. Mail all changes to *admin@eng.auburn.edu*

If a user selects all available software, then it will take a **long** time for the user to login. This is because most of the software is mounted on demand and it takes several seconds to mount each software volume. The net result is that it could take several minutes to login. Therefore, users should only select those software packages which they need to use.

Tcl and Tk: Tools for the System Administrator

Brad Morrison & Karl Lehenbauer – Paranet, Inc.

ABSTRACT

Tcl has proven itself to be very useful for adding a programmable command language to interactive programs. Tk, its offspring, has added a windowing shell, giving the user the ability to write X window applications without writing any C code. In addition, Extended Tcl has been developed, enhancing Tcl's functionality. Extended Tcl includes a general purpose shell that provides an environment for developing and executing Tcl programs. This paper will discuss the functionality provided by Tcl, Tk, and Extended Tcl, how these tools can be used by the system administrator for performing common tasks such as automating functions and adding X interfaces to existing tty-based commands and scripts. A brief comparison of Tcl and Tk to other common system administration tools is given and some examples of Tcl applications of interest to system administrators are shown.

Introduction

System administration is a task of varied roles, from ambassador to performance artist. Somewhere in between is the preferred position of toolsmith, answering the users' needs with tools rather than personal appearances. The problem with this approach lies in development time, especially user interface development. In fact, many users demand GUIs as a prerequisite for a "usable" application.

This paper presents Tcl (and its place in Tk as an embedded language) as a solution to user impatience. Tcl- and Tcl/Tk-based applications can be developed quickly, with an easily manipulated X interface. The language is readily extensible – we will also discuss Extended Tcl, a widely accepted set of extensions developed by early users of Tcl.

What is Tcl?

Tcl is a lightweight interpreter for a tool command language, hence the acronymical name. The language provides variables, procedures with argument-passing by value and by reference, familiar control constructs ('if', 'while', and 'for'), arithmetic expressions, strings, arrays, and lists. It is useful as an embedded language, or as a standalone interpreter, although its true nature is that of a library package. It is, however, a library package which just happens to implement an interpreter for a simple programming language. In particular, its pervasive use of lists and recursion lends a very LISP-like flavour, and its easy facility with strings is highly reminiscent of fondly remembered languages of yore, such as SNOBOL.

History of Tcl, Extended Tcl, Tk

Tcl was invented by John Ousterhout at the University of California, Berkeley. The original Tcl paper was presented at the 1991 Winter USENIX Conference, but according to Ousterhout, Tcl grew out of a series of projects dating back as far as 1982. These projects had a common thread: demanding user interfaces which required interactive command languages. After writing six special-purpose interpreters, he built the next one as a library that could be reused over and over. This was in 1988, and led to the first release of Tcl, in early 1989.

Extended Tcl

Since its introduction, Tcl's usage and capabilities have mushroomed. Its embeddable nature and ease of extensibility attracted many developers, who began immediately to use and extend Tcl at an furious pace. The most widely accepted extensions are those of Karl Lehenbauer and Mark Diekhans, whose contributions are called simply *Extended Tcl*. Extended Tcl added arrays and file handling to Tcl, direct access to most UNIX system and library routines, demand-loading from procedure libraries, execution tracing, and many other features. In the summer of 1991, Ousterhout merged most of these extensions into standard Tcl (version 6.0).

The Tk Toolkit

Also in 1991, Ousterhout released the Tk toolkit, a set of nine X widget classes and an X-windows event-loop interpreter which accepted Tcl commands. Tk was the first general-purpose X *interpreter*; it defined a new set of commands and data structures for creating and manipulating X widgets. Tk's procedural component was Tcl, which allowed existing Tcl applications to tack on a GUI front end. Shortly thereafter, Lehenbauer and Diekhans released support for Extended Tcl commands under Tk.

Current Tcl-based applications

expect

The first major Tcl-based application to surface was Don Libes' *expect* program, an application that

controls interactive programs which demand terminal input. *expect* uses an embedded Tcl interpreter as a high-level chat scripting language. Its impact and widespread use in a short period of time are the result of presenting a long-awaited tool, coupled with the Tcl interpreter. Aside from being easy to use, the Tcl in *expect* had already been fully developed and extensively tested.

Tk Interface Builders

Another class of Tcl applications are interface-builders for Tk. These programs, written entirely in Tk/Tcl, present an X interface for creating and modifying the graphical interface of Tk applications. The two most prominent are *BYO* and *xf*, developed by a team at the University of Wellington, New Zealand and Sven Delmas, respectively. These applications take extensive advantage of both the interpretive nature of Tcl and the meta-knowledge provided by Tk. It is trivial to query any Tk widget for its entire configuration. *BYO* and *xf* exploit this feature in saving the interface you create as executable Tcl code which is understood by Tk. This code is, of course, completely editable. Many Tk programmers use these interface builders just to create their widget definitions and placement commands, and then add supporting functions "by hand".

cute

cute (*cu*/Tcl environment) is a Tcl application which expands the functionality of the traditional UNIX program, *cu*. *cute* interprets scripts which specify actions to be taken by *cu*. By exploiting Extended Tcl's *awk*-like pattern-matching features, *cute* provides automated dialup and login to remote systems, remote command execution, and can even trick the quasi-standard UNIX *rz* and *sz* ZMODEM transfer programs into working properly on the local host.

Other Tcl Extensions

Several extension packages and applications have been written for Tcl and Tk, and most are freely redistributable. Internet users can FTP to a contributed sources archive on barkley.berkeley.edu. Applications include:

bos-1.31: Object-oriented extensions to Tcl.

cmu-snmp: SNMP interface library for Tcl, from Carnegie Mellon University.

graph-1.0: A data plotting widget.

hp-tcl-cdplay: CD-ROM player interface for HP workstations.

mxedit: A Tcl/Tk-based text editor for X-windows.

photo: A "photo" widget that can display portable pixmap (ppm) files.

rawTCP, tclTCP-1.0, tclConnect, open_socket: Four different TCP/IP interface packages for Tcl.

tcl62.dos: A port of Tcl 6.2 to MS-DOS.

tcl_curses: Adds support for the curses screen management library.

tclprof: A profiler for Tcl code.

tclvogle: A Tcl/Tk interface to VOGLE, the Very Ordinary Graphics Learning Environment.

tkWWW: A Tcl/Tk interface to the World-Wide Web, an experimental network hypertext system.

WAFE: Widget Athena Front End, a Tcl/Tk-based interface to the Athena widgets in X11R5. Can be called from shell scripts, awk and perl programs, among others, to provide an X-windows GUI.

Tcl, Tk, Extended Tcl functionality

Tcl functionality

Tcl's syntax, like most other interpreted languages, is simple and easy to read. Without rehashing Ousterhout's original syntax discussion, it is important to point out that the Tcl interpreter's recursive nature allows for nesting of evaluated expressions. Aside from the pleasant elimination of temporary variables whose only purpose is to cache intermediate results, this facility makes for extremely readable code:

```
foreach doesIt {publishes writes speaks} {
 dial [ numberOf [ personWho $doesIt ]]
}
```

However, if any intermediate results need to be stored, Tcl's `set` command conveniently returns the assigned value (see Figure 1).

```
foreach doesIt {publishes writes speaks} {
 set result [dial [ set n [numberOf [ personWho $doesIt ]]]]
 for {set i 0} {$i < 10 && $result == "BUSY"} {incr i} {
  set result [redial $n]
 }
 if {$result == "BUSY" || $result == "NOANSWER"} {
  puts $logfile "Can't reach [personWho $doesIt], Reason $result"
 }
}
```

Figure 1: Storing intermediate results

Continuing in the theme of simplicity, Tcl supports only one data type: strings. There are various rules for strings representing integers or floating-point numbers, but the interpreter does not do any conversion until the need arises. This is one of the aspects of Tcl which keeps the interpreter lightweight.

There are also two complex data structures defined in Tcl: arrays and lists. Tcl arrays are associative – the indices may be any unique string value (recall that Tcl's only data type is the string). Lists in Tcl are simply strings formatted to have a list-like structure. Tcl lists are composed of zero or more elements. Each element may be a string or another list. Further, the list-handling functions provided by Tcl return lists which are quite easily manipulated. Consider the *split* command, which breaks a string into a list of elements based on a delimiter:

```
split "dmr::201:20" ":"
```

returns the list:

```
{dmr {} 201 20}
```

Notice that the second element is an empty list.

The original Tcl command set included 29 built-in commands. The major command groups, as defined by Ousterhout, are control, variables and procedures, list manipulation, expressions, string manipulation, file manipulation, and subprocess invocation.

Programmers can extend Tcl in two different ways. You can write C code to create new commands, or you can simply create new procedures in Tcl. The latter approach is actually quite feasible, especially with the indexing facilities which allow the specification of paths to specific packages of pre-defined Tcl procedures. For example, from the original Tcl paper, here is a Tcl command that defines a recursive factorial procedure:

```
proc fac x {
 if {$x == 1} {return 1}
 return [expr {$x * [fac [expr $x-1]]}]
}
```

This `proc` command defines a new command, `fac`, to the Tcl interpreter. It may be executed from a main program file, sourced or included from an auxiliary file, or even typed in "live" in an interactive session with the interpreter. However the new command is communicated to the interpreter, it becomes available for invocation like any other Tcl command:

```
fac 4
```

will return the string "24".

Extended Tcl functionality

Most of Extended Tcl is the result of needed capability for programmers who wanted to use Tcl as a standalone language after experiencing a growing frustration with the inefficiencies and recurring difficulties in writing shell scripts, *awk* programs, etc., in the usual UNIX manner. Tcl's early proponents, after porting it to various flavors of UNIX, began extending its capabilities to meet their specific needs. As a result, Extended Tcl offers access to most UNIX system and library commands, enhanced data structures, and more efficient control structures.

UNIX Command Support

Some of the major areas lacking in early Tcl were associative arrays, file I/O, signals, and process control. Extended Tcl provides interfaces to most system calls and library routines, casting them into a Tcl context. For example, every command required to accomplish the mundane task of adding a new user is available, from *gets* to *chmod/chown/chgrp*. There are even time/date routines, with sophisticated retrieval and formatting intrinsics (see Figure 2).

Debugging and Error Handling Support

Another area which has received a lot of attention is the tracing and debugging facility. Since Tcl is an interpreted language without any intermediate code generation, syntax errors are not apparent until the code is actually executed by the interpreter. When errors do occur, Tcl offers exhaustive traceback messages. These can be caught and saved for future reference, of course, but also kept safely from the hypercritical eyes of users, while innocuous error messages can be displayed. More robust Tcl applications actually employ heuristics to detect missing or poorly configured environmental elements.

Debugging Tcl scripts once an exception has been found is merely a matter of executing a command. Every action taken, even low-level interpreter actions, can be intricately traced. An experimental Tcl debugger (Lehenbauer) supports breakpoints, single-stepping, and many other standard features associated with modern debugging tools.

File Scanning Support

Extended Tcl supports a pattern-scanning utility which works in the same manner as *awk*. Regular expressions are generated and associated with actions to be taken at pattern-matching time. These relations are then applied to entire files, or portions thereof.

```
puts stderr "That command will be implemented in exactly 30 days, at "
puts stderr [fmtclock [expr [getclock]+86400*30]]
```

Figure 2: Time & date routines

Keyed lists

The most powerful complex data structure in Extended Tcl is the keyed list (Diekhans), although, like lists, keyed lists are merely formatted strings. Briefly, keyed lists are a C structure-like method of representing data objects in hierarchical fashion, implemented simply as a list of lists. Since any element of any keyed list can be another keyed list, extremely complex data can be conveniently represented. Syntactically:

```
set person {}
keylset person {name Brad Morrison}
keylset person {hair red}
```

The variable `person` now contains:

```
{{name Brad Morrison} {hair red}}
```

The command

```
keylget $person name
```

returns

```
"Brad Morrison"
```

For a more complex definition, subfields may be specified:

```
keyldel person name
keylset person name
   {{first Brad} {last Morrison}}
```

Now the list contains

```
{{name {first Brad}
       {last Morrison}} {hair red}}
```

Subfields may be referenced with "." notation:

```
keylset person name.first "Jim"
```

changes the `$person` keyed list to

```
{{name {first Jim} {last Morrison}}
                          {hair red}}
```

Keyed lists are quite dynamic:

```
keyldel person hair
keylset person band "Doors"
```

deletes one "structure member" and adds a new one:

```
{name {{first Jim} {last Morrison}}
                        {band Doors}}
```

Finally,

```
keylget $person
```

returns the "keys":

```
name band
```

Keyed lists have value far exceeding that of simple structure emulation, however. The "keys", by which list members are retrieved, are associative, and can be assigned based on the data. This makes keyed lists uniquely qualified to manipulate RFC-822 headers, OSF stanza-format files, and so forth.

String and character manipulation

Extended Tcl added numerous string and character manipulation primitives to Tcl. Functional equivalents for most of the string routines in the C library are implemented, as are most of the *awk* functions.

The tclshell

An especially helpful aspect of Extended Tcl is its shell. Like Tcl, it will start an interpreter which reads from standard input if no file of commands is specified. This shell will accept live commands, including any UNIX command available along the current $PATH value. It offers extensive on-line help. Most importantly, it greatly facilitates modular testing by allowing procedures to be sourced and controlled by judicious use of the *$interactiveSession* variable. After the interpreter has loaded all of the necessary procedures, any scenario can be enacted. Sessions can be logged, and even automated for test suite construction.

Tk functionality

The Tk toolkit consists of a set of extensions to Tcl that provides a Tcl-programmable X-windows toolkit. Applications can be programmed for the Tk toolkit entirely in Tcl – this is usually the way small Tk applications are written. For larger applications, and for existing applications already written in C or some C-callable language where one wants to add a GUI interface to the application, Tcl interfaces to the high-level routines of the application are coded, then the "glue" between the application and the user interface is coded in Tcl.

Even with a large C application, writing the user interface in Tcl/Tk provides many advantages over coding it directly in C. Most importantly, doing the user interface in Tk results in a dramatic decrease in the time required to write it – Tk's widgets are high-level in their approach, and straightforward in their use. Also, since the application doesn't need to be relinked every time some small change is made in its user-interface, turnaround time is reduced, and even on today's powerful workstations, the size of the applications that developers are building these days are large enough that link time can still be a major factor limiting programmer productivity. Further, the quick turnaround time Tk provides gives the developer a chance to experiment with many different approaches toward the user interface, hopefully resulting in a more intuitive and easier to use product.

Finally, since Tk is interpreted, the target application is inherently programmable, without the need to license, ship or otherwise require some third party or vendor-supplied compiler to be present – savvy users can create their own interfaces, or alter the capabilities of existing ones.

Tk Widgets

Tk defines a number of widgets from which X-windows applications are created. Widgets exist to implement menus, listboxes, scrollbars, frames, labels, pushbuttons, and so forth. A summary of the available widget types appears below:

Canvas Widgets

The `canvas` command creates canvas widgets, which can be used to display and manipulate structured graphics. Canvas widgets contain items such as polygons, circles, ovals, arcs, rectangles, lines, bitmaps and text. The style of the endpoints of lines can be specified (arrowheads, butted, projecting or round), items can be stippled and the splining of curves can be controlled. Text can have an anchor position, font and color, and justification specified.

These elements can be moved, have their color changed, their 3D relief specified, and have Tcl commands associated with user actions, such as mouse clicks over the widget's elements.

Tags can be associated with one or more graphical elements and they can be manipulated en masse using these tags.

Checkbutton Widgets

The `checkbutton` command creates checkbutton widgets. Following the Motif look and feel, checkbuttons can be selected, deselected, flashed, toggled and invoked.

Clicking a checkbutton widget can toggle a Tcl variable between two different values, and/or execute a command. Changing the value of the variable from within a Tcl procedure will automatically change the state of the checkbutton.

Entry Widgets

Entry widgets are one-line text input elements, and are created by the Tk `entry` command. Entry widgets allow text to be edited using several different command options, which are typically (and by default) bound to keystrokes and mouse actions.

Entry widgets can easily be associated with a scrollbar when the text may be longer than the width of the display element.

Frame Widgets

Frame widgets (created by the Tk `frame` command) act as a spacer or container for other widgets, and these subordinate widgets can have their layout and relative positions specified by the Tk `pack` command.

Label Widgets

Label widgets are created by the `label` command. Label widgets can display text or bitmaps, and can have their 3D relief, colors, font and text defined and altered. Label widgets cannot have their text edited by the user – entry, listbox or text widgets should be used when user editing is desired.

Listbox Widgets

The `listbox` command creates a listbox widget. Listbox widgets can display a list of strings in one or more lines of text. As with other widgets, their text, colors, font and relief can be specified.

If the list is larger than the number of lines in the widget, only a subset of those lines will be displayed. Likewise if an element is wider than the widget, only a subset of its text will be displayed. It is trivial in Tk to add a vertical scrollbar to enable the user to scroll through the list, and to add a horizontal scrollbar, if desired, to scroll through elements that are longer than the widget is wide.

Capabilities are provided for inserting and deleting elements, retrieving elements from the list, selecting them, determining which ones users have selected, and so forth.

Menu Widgets

Menus are created using the Tk `menu` and `menubutton` commands. A menu widget displays zero or more one-line entries, arranged as a column. Elements can contain bitmaps or text, and a keystroke sequence can be defined as an *accelerator*. Menu items can contain checkbuttons or radio buttons, too. Menus can cascade as well, and a Tcl command is typically associated with a menu selection. As with other Tk widgets, menus can be edited extensively on the fly by the Tk/Tcl application.

Message Widgets

Message widgets are created by the `message` command. Unlike entry widgets, message widgets can contain multiple lines, and lines breaks are automatically chosen at word boundaries whenever possible. Justification can be specified as left, right or centered, and control characters that are not interpreted to do something to the widget are displayed as backslash sequences.

Radiobutton Widgets

Radiobutton widgets are created by the Tk `radiobutton` command. Radiobuttons consist of text or a bitmap, and a diamond known as a *selector*.

Radiobuttons can be selected by clicking on them with the mouse. The widget can be configured to execute a command when this happens, but typically it is configured to set a Tcl variable to a specified value. If more than one radiobutton widget is defined to set the same variable, the classic *Motif* one-button-down-at-a-time behavior occurs. If the Tcl program sets the variable to a value matching the value set by one of the radiobuttons, that radiobutton will automatically be selected, illuminating its diamond-shaped selector.

Scale Widgets

The `scale` command creates scale widgets, which are useful for setting and displaying a numeric value from a range of possible values. The value is shown graphically by the use of a *slider*, and the

range of possible values and initial value are defined when the widget is created. The value is displayed as a number underneath (or adjacent to) the slider, the lowest and highest possible values are shown at each end of the widget (they can be defined to be either horizontal or vertical sliders), and at the programmer's option, numeric tick marks can be shown as well.

Scrollbar Widgets

Scrollbar widgets, created by the `scrollbar` command, are used to provide user control over what is visible within *entry*, *text*, and *listbox* widgets. Scrollbars display a slider and two arrows. Clicking an arrow causes the text in the corresponding widget to move by one character or line (depending on whether it's a vertical or horizontal scrollbar). Dragging the slider with the mouse causes the entry or listbox widget's view to scroll correspondingly.

Text Widgets

Text widgets are created by the Tk `text` command. Text widgets are multi-line textual displays, but unlike entry and message widgets, text widgets can have font, point size, color changes, relief, underlining, and stippling occur arbitrarily within the widget. Lines break automatically, and the widget can be configured to break on character or word boundaries, or not break at all (where, when a line is wider than the width of the widget, the additional text simply is not shown).

Tags can be associated with one or more ranges of text, and the text and its appearance can be manipulated using the tags. Actions can be associated with tags, such that driving the mouse over an area of text, clicking on it, and so forth, can cause Tcl commands to be executed. A single tag can be associated with multiple ranges of text, and those ranges can be manipulated en masse using text widget configuration commands.

Marks are also supported. Marks are used for remembering specific places in the text. The text can be searched, and matches can set marks.

The text widget can be configured to allow users to edit its contents, including the expected behavior of the correct font, color and so forth being used when text is inserted. Lines rebreak as new text is entered and it becomes necessary.

There are many default bindings, and of course additional actions can be created and associated with other Tk widgets to provide expanded editing capabilities.

The text widget is so powerful that the authors believe it can be the basis of a full-featured word processor.

IPC with the send command of Tk

Tk provides a mechanism for communicating with other Tcl interpreters. Any Tcl command can be sent to another Tk interpreter which is using the same X server via the `send` command. The result is returned as if the command had executed in the local Tk process. This presents an interesting question, however: How to send an "exit" command? The sender always waits for the returned result, but the remote interpreter shuts down immediately upon receiving the command, and so cannot acknowledge the receipt to the sender. The solution is interesting: The command sent must include the Tk `after` command, which schedules the future execution of commands. If the local interpreter sends the command

```
after 5 destroy .
```

the remote interpreter can reply before killing itself. In fact, every application should implement some sort of "terminate" procedure, to take care of any cleanup activities which it encapsulates.

wish: A Windowing Shell

By default, building Tk and linking it with Tcl produces a program called "wish" which stands for "windowing shell." This shell invokes a Tcl interpreter, then sets up all of the C-based, extended commands added to the interpreter. After initialization is complete, the Tk X-windows event loop takes over. This is a more or less typical event loop. The program waits until events such as mouse movements, button clicks, keyboard input, etc, occur, and when they do, it swings into action, dispatching Tcl code to be executed in response to the events. This code then performs the requested tasks, as required, possibly interacting with the user interface in additional ways.

Comparing Tcl to other Tools

Tcl and the Standard Shells (sh, csh)

The good news is that both the Bourne shell and the C shell are still distributed with every UNIX system (although the latter's implementation is not always

```
x = 1
while [ $x <= 100 ]
do
    x='expr $x + 1'
done
```

```
set x 1
while {$x <= 100} {

    incr x
}
```

Figure 3: Tcl -vs- Bourne Shell

robust or thorough). The ultimate in portability is still a well-crafted shell script.

The bad news is that every system administrator has experienced the frustration of encountering the shells' limitations. To paraphrase a great Zen master, shell programming is, at best, a ladder too short to reach nirvana, and too often, a ball and chain holding us fast to the confines of the material world. Consider the ubiquitous and mundane task of incrementing a counter. Compare the Bourne shell's method with Tcl's as shown in Figure 3. Although the commands seem to map exactly one-to-one, the crucial difference is that the Bourne shell script will generate a subprocesses for each `expr` call, forking and exec'ing 100 times. Tcl's `expr` is built-in.

On the other hand, most C shell implementations have long since internalized arithmetic expressions:

```
set x = 1
while ( $x <= 100 )
      @ x += 1
end
```

Yet neither shell handles field extraction gracefully. Witness two methods (from a Bourne shell script in use as you read this!) of obtaining the current hour and minute fields from a string `$time`, formatted simply as "hh:mm" (see Figure 4).
Which is the lesser of the two evils – the gnarly `expr` expression, or invoking *awk* just to dig out one field? Compare to the Tcl code, which does the same job more efficiently:

```
set timeList [split $time ":"]
set hours [lindex $timeList 0]
set minutes [lindex $timeList 1]
```

Tcl and awk

Awk is another tool which can always be counted on to be present on a standard UNIX system, although its capabilities can vary. It was the first general-purpose filter, and it is unmatched for its ability to select and operate on fields in every line of a given file or files. Yet *awk*'s unique way of treating its input is also a drawback. An annoyance of *awk* it its implicit pattern matching at the "outside" of the program, which often dictates inversion of the procedural approach the programmer would normally take. Further, scanning for subordinate patterns requires the use of extraneous "state" variables. Extended Tcl's notion of *scan contexts* elegantly solves both problems, and its hybrid Boyer-Moore and regular expression string searching yields excellent performance. Further, *awk*'s limitations in the area of the user interface practically restrict it to the filtering of data.

Tcl and perl

Much, if not all of the programming done by system administrators boils down to data processing. We gather information from a variety of sources and present it to users, to the system, even to ourselves. Clearly, strings are the major data type in this type of programming, which is what drives us to higher level languages than C. The shells and *awk* are sufficient for small programs, but even moderately complex tasks are harder to develop – and to maintain. To many discriminating system administrators, *perl* is the undisputed champion of parsing and report generation – it's even been used to *read* reports!

Yet *perl*'s learning curve is among the steepest of any tool available to the system administrator. Like Tcl, its functionality set is rich, but the language has an arcane syntax, and is difficult to master. *perl* programs can be hard to understand and maintain. The implicit operands which lend *perl* its concise and powerful magic may boost the programmer's confidence, but may cost more in the long run.

Tcl also requires a time investment before really useful programs can be produced. The main difference seems to be that Tcl provides more of a framework, a scaffolding which can be filled in to suit the application, with custom programmer extensions, or selected parts of Tcl libraries, demand-loaded as the need arises. The tradeoffs between Tcl and *perl* stack up quite evenly, until the Tk interface is considered.[1] Suddenly, user input is less a matter of "What was just typed?" and more concerned with which buttons the user just pressed. In the event-driven paradigm, programming is far less procedural. Most input decisions are laid out at widget creation time, although they may be changed – even while the application is running. Tcl achieves its power with a simple syntax and a rich set of functions – much like C.

Tk and Xlib/Xt

As mentioned above, when string manipulation is at its most intense, higher level programming than C is called for. The tradeoff between development time and the edit-compile-link-test-edit cycle, versus Tk/Tcl's rapid prototyping environment and its excellent string handling dictate the use of Tk,

[1] Although there is a way to add X interfaces to *perl*, it is Tcl-based.

```
hours='expr "$time" : "+*[0-9]1,2).*"'
minutes='echo $time | awk -F: '{ print $2 }''
```

Figure 4: Obtaining values for time expressions

unless execution speed is of absolute importance. Tk presents well-defined widgets to the programmer who needs to develop quickly and accurately, whereas Xlib or Xt offers more control, but also requires more responsibility, and considerably more effort.

Tcl and the System Administrator

If a poor workman blames his tools, then a clever sysadmin exploits his – especially the ones he creates. Tcl provides a comfortable environment for crafting tools to manipulate systems and their configurations.

Automating tasks

Even the fastest touch typists grow tired of pressing the same old redundant key sequences. Tcl lends itself very well to the task of generalizing mundane, repetitive, yet sensitive jobs.

Monitoring systems and networks

Tcl's full-featured command set suits it well for the performance of monitoring duties. For local monitoring, Tcl can crack the output of *ps* and *pstat*, et al. For remote monitoring and gathering network statistics, Tcl has extensions which manipulate sockets and even exploit the Simple Network Management Protocol.

Customizing the environment

Because the Tcl interpreter provides only a minimal set of defaults, the sysadmin can exercise a great deal of control over the Tcl environment. The main runtime initialization file is a Tcl script, allowing the setup to be as simple or extensive as necessary. Everything from environment variables to auto-loaded Tcl procedure libraries can be controlled from one central location.

Tcl for Heterogeneous System Administration

Because Tcl has been ported to every major UNIX platform, all of the compatibility issues are covered as soon as the installation is complete. Tcl applications are extremely portable, with no modification required between platforms.

System-wide defaults via TclInit.Tcl

Any customization necessary beyond the operating system's constraints, such as local policy on a particular subnet, or any other such specialization, can be easily applied via the TclInit.tcl script. This is, in effect, the /etc/profile for Tcl. Conventions can be established or circumvented, as need dictates.

Hiding differences with Tcl auto-loading

Another level of incompatibility may be present above the operating system itself. Different flavors of UNIX, especially the neapolitan BSD-SysV hybrids, can cause problems when one directory may be in the right place, but another is not. What looks like Berkeley-out-of-the-box may have some lurking System V components. Instead of writing all of this intelligence into each and every application, the Tcl procedure libraries which are auto-loaded or demand-loaded can be customized to smooth these differences, for a more data-encapsulated approach. This is particularly useful on platforms which comply with UNIX standards rather than actually implementing the operating system, such as AIX.

GUIs for scripts and system facilities

Of course, the most instantly gratifying dimension of Tcl lies in using it to drive Tk. Rather than re-writing existing, tried and true applications, sometimes it's easier (and quicker) to simply build on a Tk front end. For example, a long-standing application which monitors a WAN happened to record its status in a group of log files. When a graphical interface was requested, it was a simple matter of representing the WAN in a canvas widget with push-buttons for the sites and using color codes to reflect the status. Clicking the buttons yielded further information, all gleaned from the log file, but presented in a more discriminating fashion. Best of all, the front end was developed in an afternoon, by a Tcl developer who only knew the meaning of the log file entries.

Practical Applications of Tcl

Application launching with Tcl

The Magicbutton application

The first Paranet system administrator to arrive at the Dallas offices of a large client company found, of course, many problems in need of attention. None was more compelling that what their users were having to go through to launch CPU intensive applications.

Our administrator was amazed to find the perimeter of their monitors dotted with Post-It notes containing instructions for starting various applications. The typical method of launching an application was to telnet to an IP address (they didn't have any Internet name services beyond /etc/hosts), log in, set their display environment variable to the IP address of their display server (of course the remote machine didn't run NIS or DNS either), *cd* to a long directory path, then launch the application by a long, explicit pathname.

The sysadmin quickly brought up Tcl and Tk, along with the extensions provided by Extended Tcl, and began work on MagicButton, an application to enable users to launch applications by pushing buttons in an X-window. Because several of the remote programs expected to be able to open /dev/tty, simple remote execution based on *rsh* was inadequate. He added *expect* to handle the task of logging into and managing the interaction with the remote session. With Tcl, Tk, and *expect*, the program was mostly working the first day, ported readily to their HP, DEC and IBM workstations, and it was trivial to

do things like substituting bitmaps for text on the pushbuttons.

Two days later, with several users happily running MagicButton, our sysadmin left the site with his reputation as a god cemented there forever. Since then MagicButton has spread to the company's operating groups all over the world.

Hypertext with Tcl

A Hypertext man page application

After obtaining a hypertext widget extension for Tk from George Howlett, we decided that, although it had terrific functionality,[2] it needed to be more general. One of the most daunting aspects about hypertext is the incredible time investment necessary to create a document and define all of the links. What if a hypertext system included a processor to figure out the links on the fly? The online man pages seemed ideal – there are a lot of them, and they're loaded with keywords for possible "hot buttons".

The first step was to perfect the ability to spot all of the possible links. The two main clues in a man page are the underlined words – which really are just words whose letters are surrounded by underscore and backspace characters – and the explicit keywords, like the ones under the "SEE ALSO" heading. Once these were parsed, they were handed to the hypertext widget, along with instructions on how to find the associated man page. Italic and boldface elements were extracted and displayed as well. Effectively, the act of displaying a man page under the hypertext application set up all of the links on the fly, allowing users to view the manual pages as a large, integrated hypertext document.

System Management with Tcl

A ttytabs Editor

This project began with an idea for a tool to display the state of the *ttytabs* file, using the Tk toolkit. It became evident that changing the state of the file would be a logical and straightforward extension. Since all but the `id` field of the file's records consist of a fixed set of values, radio buttons are the obvious choice for field value selection. Additional pushbuttons move backward and forward in the file, displaying each record in turn. Due to the variable length of the file's records, however, the entire file must be rewritten, even if only one record is changed.

Conclusion

Extended Tcl has proven to be an effective tool for writing programs that otherwise would have been written as a combination of shell scripts, *awk* and *sed* programs, and so forth.

Using the Tk toolkit we have created X-windows applications from simple to sophisticated, programming them interactively, with the convenience and short turnaround time of shell programming, but with the higher performance and more rational structure of Tcl. Furthermore, with Tcl there isn't the need to drop into other languages to work around various limitations in each tool's capabilities.

Tcl and Tk's demonstrated portability to a wide range of systems, including Sun, HP, Apollo, IBM, DEC, Silicon Graphics, Cray, and SCO UNIX, spanning a range of performance from the Intel 386 to the Cray Y/MP, is particularly important in today's heterogeneous environments.

For system administrators, Tcl and Tk provides an opportunity to quickly create scripts with GUI interfaces, simplifying user's tasks and reducing the amount of personal attention required of the sysadmin, giving them more time to read news, prepare conference papers for publication, and perform other important duties.

Acknowledgments

Thanks to John Ousterhout of the University of California at Berkeley for inventing Tcl and Tk, and for supplying us with information on the history and origins of Tcl.

Thanks to Mark Diekhans of the Santa Cruz Operation for his work on Extended Tcl, and the input he provided during the preparation of this paper.

References

John Ousterhout, "Tcl: An Embeddable Command Language", *Proceedings of the Winter 1990 USENIX Conference*, January, 1990.

Don Libes, "expect: Curing Those Uncontrollable Fits of Interaction", *Proceedings of the Summer 1990 USENIX Conference*, June 1990.

John Ousterhout, "Tk: An X11 Toolkit Based on the Tcl Language", *Proceedings of the Winter 1991 USENIX Conference*, January, 1990.

Larry Wall and Randal L. Schwartz, *Programing perl*, Sebastopol, CA: O'Reilly & Associates, Inc., 1991.

Karl Lehenbauer & Mark Diekhans, "Extended Tcl – Extended command set for Tcl 6.2", unpublished manual page, January, 1992.

Sven Delmas, "Announcing XF – an interface builder for Tcl/Tk", Usenet News Article ID *GARFIELD.92Jun15210119@avalanche.cs.tu-berlin.de*, June 1992. Email address: garfield@cs.tu-berlin.de

Karl Lehenbauer, "A source level debugger for Extended Tcl", Usenet News Article ID *1992Jan03.220658.22059@NeoSoft.com*, January, 1992.

[2]Text items as Tk pushbuttons, which of course were fully configurable for launching any command.

Mark Diekhans, "Tcl performance profiler", Usenet News Article ID *1992Aug02.060921.5504@NeoSoft.com* August, 1992. Email address: markd@grizzly.com

George A. Howlett, "Version 1.0 of xygraph, htext widgets ...", Usenet News Article ID *1992Jun15.142329.8170@cbnewsm.cb.att.com* June, 1992. Email: george.howlett@att.com

Andrew Robinson, James Noble, Peter Wood, Roanne Steele, Alexander Leadbeater, Alan Young, and Paul Scheffer, "BYO – A User Interface Builder for Tcl/Tk", Usenet News Article ID *1992Mar6.025636.21564@comp.vuw.ac.nz* March, 1992. Email address: byo@comp.vuw.ac.nz

Author Information

Brad Morrison graduated from the University of Texas at San Antonio in 1987 with a BS-CS after repeated exposure to uncontrolled bouts of UNIX system administration at Southwestern Bell. After graduation, he moved to Houston, doing UNIX systems administration, systems programming, and applications development for Ferranti International Controls Corporation and the SYSCO Corporation. Brad is currently a systems and network administrator with Paranet, Inc., working on Tcl/Tk products. Mail to Paranet, Inc., 1743 Stebbins Drive, Houston, TX, 77043; E-mail to morrison@paranet.com.

Karl Lehenbauer studied Computer Science at Indiana University from 1977 to 1981, and was the first UNIX programmer on campus, bringing up and hacking on IU's first UNIX systems. Since then he has been a contract programmer, doing oil and gas programming and design for Hydril, realtime power systems programming at Utah Power and Light and Ferranti International Controls, programmed aircraft instruments for General Electric, worked as a consultant on the Advanced Computing Environment for the Santa Cruz Operation, and now consults with Paranet, doing Tcl/Tk programming and working on special projects. Mail to Paranet, Inc., 1743 Stebbins Drive, Houston, TX, 77043; E-mail to karl@neosoft.com.

Concurrent Network Management with a Distributed Management Tool

R. Lehman, G. Carpenter, & N. Hien – IBM T. J. Watson Research Center

ABSTRACT

As distributed computing has become more prevalent, the need to effectively monitor and manage computer networks has grown in importance. In the the TCP/IP world, this has meant monitoring routers, gateways, hubs and other devices using SNMP and other protocols. However, network management stations have tended to be single-use, turn-key applications that lack scalability. In addition, monitoring and measurement have extended beyond the level of routers and gateways to end-user workstations where system administrators are using SNMP to keep track of traditional system management functions using the existing network management framework.

Most existing monitoring and management tools do not scale when the monitored network may include hundreds of routers and thousands of workstations. For large, complex networks, it is impractical to have a central monitoring and data collection point that generates all management queries, stores results, and processes alerts and traps.

To monitor and manage the TCP/IP network at the IBM Thomas J. Watson Research Center, we are currently using the DRAGONS[1] Data Engine, a distributed, object-oriented, run-time environment which supports multithreaded tasks. In this paper, we show how the latencies in polling and alert notification, which can occur in large networks with a central management station, can be reduced by employing multiple Data Engines on multiple hosts to perform management tasks simultaneously. While the DRAGONS Data Engine is a general purpose, run-time environment, it is particularly well-suited to network monitoring, since the problem of polling a large network can be decomposed into small, light-weight queries which map onto the Data Engine's multithreaded environment quite well.

Introduction

The efficient monitoring and management of the components that comprise networks is obviously an important function in any distributed computing environment. In general, if the network fails, the computing environment is rendered useless. As networks have grown in size and importance, network management systems have become critical pieces of software. Unfortunately, network management systems have tended to be single-use, turn-key applications that lack both the flexibility and scalability to deal with situations where the monitored network may include hundreds of routers and thousands of workstations. For such complex networks, there comes a point where it is impractical, if not impossible, to have a central monitoring and data collection point that generates all management queries, stores results, and processes alerts and traps.

One solution to the scaling problem is to distribute the management tasks among different processes in the existing distributed computing environment. This can be attractive if an infrastructure exists that facilitates the development of distributed applications. The DRAGONS Data Engine, developed at IBM Research, provides such an infrastructure and this is the approach we have taken at the IBM T. J. Watson Research Center.

The IBM T. J. Watson Research Center Network

The TCP/IP network at the IBM T.J. Watson Research Center extends to eleven buildings at five sites in Westchester County, New York. The sites are interconnected by either leased T1 lines or fiber. There are over 3500 active TCP/IP systems (AIX, SunOS, DOS, OS/2, VM and MVS) spread across the sites, interconnected by over 20 IP routers. The LAN topology includes token ring, ethernet, and FDDI networks. Besides the local networking infrastructure, links to other sites in North America, South America, Europe and Asia are also monitored.

In addition to the geographically distributed devices comprising the IP networking infrastructure, a large number of local compute- and file servers are monitored and their respective owners are notified (either in real-time or via nightly reports) of outages associated with those systems.

Unfortunately, as more and more routers, workstations and servers were added to the set of devices to be monitored, our existing monitoring methodology did not scale satisfactorily. The

[1] DRAGONS is an acronym for Distributed Reliable Architecture Governing Over Networks & Systems

polling cycle latency (the amount of time it takes to probe and verify the status of each monitored device) was increasing to such an extent that it was becoming infeasible to notify system and network administrators in real-time of failures of important components.

To reduce the polling cycle latency time, we began to use the DRAGONS Data Engine to perform network monitoring and management. It allows us to have a coherent management system that enables the transparent distribution of the workload among a number of systems in the network. However, since our experience with a newly deployed distributed tool like the Data Engine is limited, it was not obvious what was the most efficient workload mix (number of Data Engines and number of threads per host) given our networking infrastructure.

Short Overview of the DRAGONS Data Engine

While the focus of this paper is not on the DRAGONS Data Engine, a short, high-level overview of the environment is useful.

The DRAGONS Data Engine is a distributed, multi-threaded, object-oriented, application development environment. The Data Engine core provides three fundamental abstractions to applications: classes, objects, and threads.

Classes provide a means of organizing data and associated operations in a well-defined fashion. A Data Engine class definition defines the instance variables for each object of a class as well as the methods defined for objects of the class. Methods are similar to functions defined on a specific data type: they are reentrant code bodies that are to be executed as a thread. In the object model defined by the Data Engine, methods are the only external interface to an object. The DRAGONS Data Engine supports multiple inheritance.

An object has several characteristics: it has a unique name (its object ID), it is a member of a class, and it has a state. The state of an object is determined by the values of its instance variables.

The underlying architecture providing the Data Engine run-time environment implements tagged variables that identify the type of variables to the run-time environment. This information is heavily exploited by the free storage reclamation mechanisms and often by applications. Several primitive types are directly supported. These include abstractions for integers, floating points, octet strings, object IDs and native language messages (used to directly support internationalization and locally customizable messages). Composite types such as sparse arrays, associative arrays and sets are also directly supported by the run-time environment. All data can be freely passed between heterogeneous machine architectures.

Method invocations manipulate an object's state. The Data Engine run-time environment creates a new thread for each method invocation. While this may initially appear to be prohibitively expensive, overhead has been kept low and future optimizations may improve performance even more. Using a simple RPC benchmark that is intended to measure overhead of the environment, on an IBM RISC System/6000 Model 560, a Data Engine can process over 2600 method invocations per second and more than 3900 context switches per second. These numbers correspond to over 1300 threads created per second (not all method invocations result in new threads being created because either the method body was null or the invocation was addressed to a thread) and over 650 RPC-style object interactions per second.

Exploiting the Distributed Capabilities of a DRAGONS Data Engine

A basic DRAGONS Data Engine with no local customization includes built-in classes ranging from fundamental, low-level classes that provide access to primitive operating system facilities like timers, files, TCP and UDP sockets to those that implement application support services like interfaces to SNMP agents, a notification service that can display messages on a user's display, send e-mail or drive a pager system, a plotting program to display graphs on a user's terminal or a job scheduler.

DRAGONS Data Engines can be operated as standalone systems; however, by exploiting their support for transparently distributed operation, several interesting possibilities become available. These include utilizing multiple CPUs to process a job, and fault-tolerant operation. The focus of this paper is on exploring the use of multiple CPUs.

The JobController is an example of a Data Engine class that we developed. An object of this class can be instantiated to provide a distributed scheduler function for a job. When the object is created, the maximum number of threads to be created on a particular host for the job is specified, along with the maximum number of hosts that can participate in the computation. If fewer hosts are available than the number specified, the number available is used. By using such an object, an application can be written so that it works when running on a single CPU, but it can automatically take advantage of additional processors when they are made available. The complete implementation of the JobController class, incidentally, is not a complicated piece of code.

The JobController class is used by another sample Data Engine application that polls networks of SNMP hosts. The complete source code for this application appears in Appendix A. A slightly modified version was used to obtain the timing results reported in this paper.

Optimal Distribution of the Polling Workload

While the network management tasks may be distributed over multiple Data Engines, less than optimal performance will be realized unless the scheduling is tuned. This involves determining the number of threads that a given Data Engine should devote to a particular management job, as well as the optimal number of CPUs to be used. Our initial premises were:

- As the latency in the network increases, it is more useful to have increasing numbers of threads that can perform computation while waiting for slow network responses.
- Conversely, as the latency in the network drops, it is more appropriate to have decreasing numbers of threads.
- As the amount of computation required increases, it is more appropriate to decrease the number of threads. As an extreme case, a program that does nothing but computation (no I/O) will run fastest on a dedicated machine which does no timesharing.
- Conversely, if the jobs perform large amounts of I/O, then having more threads allows for the otherwise wasted processor idle time to be used.

With respect to the optimal number of Data Engines, we worked from the following premises:

- A speed-up would only be realized when the unit of work to be distributed was greater than the effort involved in scheduling the work to be done on a remote CPU.
- To achieve as close to a linear speedup as possible for each CPU added to the processor pool, the unit of work to be distributed would have to be constructed in such as fashion as to minimize the amount of interaction with the master object responsible for the overall completion of the job.

Experimental Results in a Local Area Network

To determine the best mix of Data Engines and threads per active Data Engine, a set of experiments was developed to measure polling cycle times while varying the numbers of Data Engines and threads-per-host. This was a simple experiment to setup since the JobController class within the Data Engine performs scheduling of tasks across multiple Data Engines, and varying the number of Data Engines and threads is accomplished without requiring changes or recompilation of the measurement code. We identify the Data Engine running the scheduler as the master Data Engine and any extra Data Engines are called slaves.

As mentioned earlier, the IBM T. J. Watson Research Center complex has IP connectivity to other sites throughout the world, and links to these sites are monitored; however, the target topology measured in the first experiment is a subset intended to be representative of a LAN (vs. a WAN) environment. The subset used for the experiment is an "extended" LAN environment, in which the majority of measured machines are less than three hops away. Only a small fraction of the monitored hosts are reached by a "slow" (T1) serial link. The average round trip time (as measured by ping) between the workstations running the Data Engine and the monitored machines is approximately 15 ms.

A total of 238 hosts were measured, some of which were IP routers with multiple interfaces where each interface on the router was probed on each pass by retrieving relevant MIB variables (see the previously presented source code for details). The majority of the hosts (201 out of the 238) were simply tested by sending an ICMP ECHO request. To poll the 238 hosts, a total of 548 request/response interactions were performed.

The methodology for the experiment was simple: a number of trials were performed, increasing the number of threads per host from one to 30 threads. At each threads-per-host level, trials were performed with an increasing number of processors in the processor pool. Processor pool sizes of one, two and four processors were used. The experimental runs were performed over a number of nights to normalize for bursts of traffic, which in the target environment are less numerous outside of normal working hours, and the average time for each threads-per-Data Engine/processor total combination was plotted. Approximately 100 trials were performed for each data point.

In the best case scenario (four Data Engines each running with seven threads), the polling cycle time was approximately 30 seconds. Given that 238 hosts were polled, this means that the average time need to poll each host in this configuration was 0.13 second. The system was processing a little over 18 request/response interactions per second with the monitored devices. This contrasts to the worst-case performance measured (one thread running on a single Data Engine), in which the average polling cycle time was 104 seconds, yielding a cost of 0.43 second per host. At this rate, the system was processing a little over 5 request/response interactions per second with the monitored devices.

The results of these experiments can be seen in Figure 1. The benefit of multithreading in this application is obvious. A single thread running on a single host takes nearly twice as long to complete the polling cycle as two threads running on a single Data Engine. Obviously, the interesting question is: When does adding Data Engines, and increasing the number of threads per host, stop improving and start degarding polling cycle time performance?

It is apparent that 15 simultaneous threads per host constitute an effective upperbound, after which

performance suffers. The value of this upperbound is due, in part, to the limited number of parameters altered in this experiment. As a consequence, the job scheduler allocates the same amount of work for the master CPU as it does for slave CPUs. This creates contention within the master Data Engine because the dequeued work units compete for timeslices along with the distributed scheduling functions.

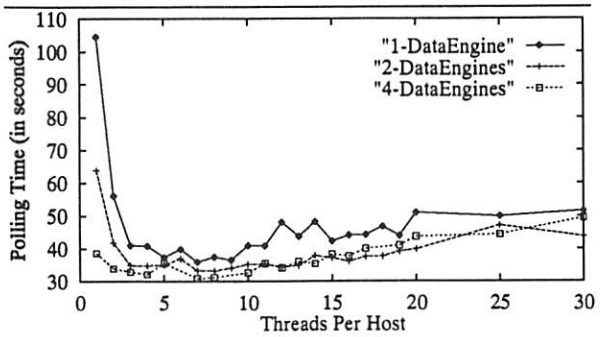


Figure 1: Polling times

In this trial, a single Data Engine does not perform as well as a pool of two or four Data Engines, but the difference between two and four Data Engines is less clear. The graphs illustrate that the actual differences in polling cycle latencies with varying numbers of Data Engines are small, and are only unreasonably large with one or two threads per host. This can explained by the fact that in the target topology, the vast majority of the distributed work units do a trivial amount of work (they merely generate an ICMP ECHO request), and thus the cost of actually sending a task to a remote Data Engine is essentially the same as processing it locally. If the work units delegated to slave CPUs were more computationally-intensive, then we would expect to see larger differences.

To prove this premise, another set of trials was performed using a set of topology data that contained only SNMP-based devices. The results of these trials are shown in Figure 2. The percent reduction in per-host processing time was compared between the original topology and the SNMP-only topology. Using 4 CPUs instead of only one provided a 14% decrease in runtime with the original topology, and a 60% decrease in runtime with the SNMP-only topology. These results support the premise.

In general, it appears that for the original topology of monitored devices, the optimal number of threads-per-processor seems to be between seven and ten. While the use of additional Data Engines does yield better performance, the improvement is not sufficiently large to motivate the use of multiple Data Engines purely for the sake of reducing polling cycle latencies in this particular network. This lack of significant improvement is due to the fact that the topology in question contains an inordinate number of devices which are probed using merely ICMP ECHO requests. When the unit of work to be distributed is more computationally-intensive, the value of using additional Data Engines becomes more apparent.

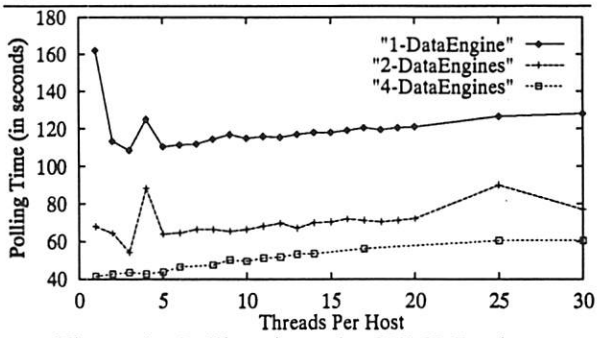


Figure 2: Polling times for SNMP Devices

As noted earlier, performance begins to degrade after about 15 threads per host simply because the master Data Engine becomes CPU-bound. The goal is to create a Data Engine configuration where we have just-in-time monitoring: each Data Engine should run as close to capacity as possible. Multithreading is used to permit a Data Engine to spend time generating queries while waiting for replies to previous queries, and in an ideal configuration, the generation of new queries would end just as replies to previous queries were being returned. The problem exposed by these experimental results is that the master Data Engine becomes a a bottleneck because the distributed scheduler loads the master Data Engine with the same number of work units as the slaves. Better performance in the multi-user case might be achieved by increasing the number of work units offloaded to the slaves and decreasing the number of work units assigned to the master Data Engine.

To gain some insight into the effect of the default equal bias-scheduling policy, a third set of trials was performed with the scheduler instructed to avoid delegating any work units to the master Data Engine. In such a scenario, the master Data Engine is responsible only for scheduling and the slaves are responsible for actually making progress on the job. The results obtained demonstrated that the mechanics of distributing work units among multiple hosts becomes the bottleneck.

A consistent lower bound was observed, regardless of the number of hosts made available, indicating that the master Data Engine was 100% saturated and the slave CPUs were processing work as quickly as it could be delegated. As a result, we have determined that to further improve performance, we will need to invest more effort in the implementation of the JobController task. Two new approaches present themselves. One is to attempt to reduce the number of method invocations per monitored device that must cross host boundaries. The current total is four

and it should be straightforward to reduce this to two method invocations. The second approach, that is not mutually exclusive with the first, is to attempt to distribute the scheduling function itself. This is not as straightforward as a static (pre-calculated) distribution of work units that does not take into account effects introduced by lost packets and unresponsive hosts.

Experimental Results in a Wide Area Network

The results obtained from a LAN environment with low network latency were in line with our expectations, but the effect of multiple DRAGONS Data Engines was not dramatic enough to compel their use. We had run trials to confirm that multiple Data Engines had a greater effect when the distributed work units were more computationally-intensive, but we felt it would be useful to explore the performance of Data Engines in a wide area (T1 and T3 based) TCP/IP network backbone.

This set of trials was conducted on the ANSnet backbone network. The characteristics of this network are different from the Watson LAN environment since it is composed primarily of serial links. In addition, the monitored devices are generally further away in terms of numbers of hops. As the number of hops increases, a corresponding increase in round-trip times is measurable. The average round trip time between the measurement points and monitored hosts in this network is 52 ms, almost 3.5 times longer than in the Watson IP LAN.

The same code and experimental methodology was used for this target environment with two modifications: a maximum of two Data Engines were used, thus eliminating the case of four processors in the processor pool which was tested against the Watson Research Center LAN; and the processors were slower. The results are shown in Figure 3.

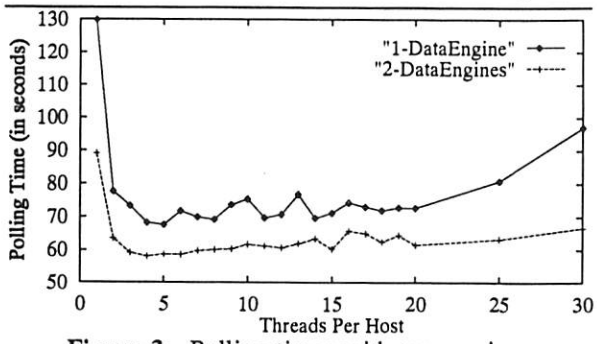


Figure 3: Polling times with two engines

In this environment, the addition of a second Data Engine had an obvious effect on the polling cycle time. Between 10 and 20 threads, the second Data Engine reduced the polling cycle time by 10 seconds. In the best case configuration (two Data Engines each running six threads) the average polling cycle time was about 60 seconds. For a total of 98 monitored hosts, the average polling time per host was 0.61 second. This contrasted with the worst case scenario (one thread running on one host) where the average polling cycle time was 129 seconds, or 1.31 seconds per host.

In both the one and two Data Engine cases, performance began to degrade noticeably at about 20 threads per host. Again, the bottleneck of the distribution of work units limited throughput.

Comparing the LAN and WAN results

In the ANSnet backbone, the probability of packet loss is greater than that of the Watson LAN. We see the benefit of using more than one Data Engine to compensate for the loss of throughput when a thread is held up waiting for a retransmission to take place. In the single processor case, a packet loss event temporarily drops the system back to making progress with T - 1 threads, whereas in the two processor case 2 * T - 1 threads remain active.

Better performance is achieved with a fewer number of threads in the case of the ANSnet backbone as contrasted with the Watson LAN. This is as expected: there are no trivial units of work being distributed because all of the hosts being polled are SNMP-based. Since the computation involved in dumping the tables of a router is greater than in generating an ICMP ECHO request, fewer units of work are required to saturate a given processor.

Summary and Future Directions

- The use of multi-threading is a very effective technique for reducing polling latencies.
- The use of multiple Data Engines also contributes to a reduction in polling latencies.
- Multiple Data Engines are more effective when the unit of work to be distributed is computationally-intensive; otherwise, the effort involved in delegating the work to a remote processor far outweighs the savings gained by offloading the work.
- Our simplistic distributed scheduling application has become a bottleneck for increased performance.

The issue of efficient workload distribution needs to be addressed with several avenues of exploration not discussed in this paper.

- The mechanism that starts and controls work units on remote hosts can be improved. This currently involves 4 method invocations which must cross host boundaries.
- An alternative approach to the current dynamic scheduler is to partition the topology database into fixed set of hosts and delegate these to slave CPUs. The slave CPUs would send one message back to the master informing it of the status information they had collected. On the negative side, such a static

scheduling approach makes it more difficult to add or remove CPUs from the processor pool while the job is in progress.

- Data Engines already understand the concept of delegating requests for certain networks or hosts to authoritative Data Engines. This is used, for example, to automatically route requests for external networks through secure IP gateways, enabling transparent SNMP access access to external networks. It can also be used in a wide area network environment to route processing of requests closer to their ultimate destination, thus eliminating the latency involved in repeated interactions with a very remote site.

Availability

DRAGONS software can be licensed for the IBM RISC System/6000 and Sun (SPARC-based) platforms. Inquiries can be sent to dragons@watson.ibm.com or the authors.

Author Information

Robert Lehman is a Staff Programmer in the Watson Networking Systems group at IBM Research. He is responsible for management and monitoring tools for the Watson IP network. He can be reached via US Mail at IBM T. J. Watson Research Center, PO Box 704, Yorktown Heights, NY 10598, and via networked electronic mail at rlehman@watson.ibm.com.

Geoffrey C. Carpenter is an Advisory Programmer at IBM Research. He works in the Advanced Information Technology Group (Department of Computing Systems) and is the author of XGMON, an SNMP manager for TCP/IP networks, and is the key developer of DRAGONS. He can be reached via US Mail at IBM T. J. Watson Research Center, PO Box 218, Yorktown Heights, NY 10598, and via electronic mail at gcc@watson.ibm.com.

Nguyen C. Hien is with IBM Research, presently the Manager of Automation Systems in the Advanced Information Technology Group (Department of Computing Systems). He has development responsibility for Systems and Network Management tools, in particular XGMON, an SNMP manager for TCP/IP networks, and DRAGONS, an object-oriented environment for distributed applications. He can be reached via US Mail at IBM T. J. Watson Research Center, PO Box 218, Yorktown Heights, NY 10598, and via electronic mail at hien@watson.ibm.com.

Appendix A: Polling Code

```
include "DEinterfaces.oog_h"
include "DEtype_util.oog_h"

enum states {
        IN_PROGRESS, COMPLETED, NO_RESPONSE, NO_SNMP_RESPONSE
};

class SNMP_PollNetwork {
        oid             controller;
        int             startTime;
        int             jobsCreated;
        int             jobsDone;
        int             tc, hc;
        array(string)   community;
        array(int)      addresses;
        set(oid)        hosts;
} inherits from Object;

SNMP_PollNetwork:create(int threadCount, int hostCount)
{
        int             t, h;

        t = 10; h = 1;

        if (argc >= 1) t = threadCount;
        if (argc == 2) h = hostCount;

        tc = t; hc = h;

        send "schedule"(tc, hc) to thisObject from nil;
}

SNMP_PollNetwork:schedule(int threadCount, int hostCount)
{
```

```
        int             i;
        oid             host;

        write_cout("Starting to schedule, t = ", tc, " h = ", hc, "\n");

        if (controller == nil)
                controller = send "createObject"("JobController", tc, hc)
                                to RootObject;
        if (hosts == nil)
                hosts = send "allInstances"("SNMPhostInfo") to RootObject;

        startTime = localRelativeTime();

        i = 0;
        for host in hosts {
                i = i + 1;
                if (community[i] == nil)
                        community[i] = send "getReadCommunity" to host;
                if (addresses[i] == nil)
                        addresses[i] = send "getAddresses" to host;
                send "queueJob"("SNMP_PollHost", thisObject, host,
                        community[i], addresses[i]) to controller;
                jobsCreated += 1;
        }
}

SNMP_PollNetwork:delete()
{
        send "doneWithJob" to controller;
}

SNMP_PollNetwork:doneWithHost(oid obj, int success)
{
        int             endTime;

        jobsDone += 1;
        if (jobsDone == jobsCreated) {  // all done...

                endTime = localRelativeTime();
                write_cout("All ", jobsDone, " hosts polled in ",
                        endTime - startTime, " seconds!\n");
                jobsDone = 0;
                jobsCreated = 0;
                send "schedule"(tc, hc) to thisObject from nil;
        }
}

// poll a host
class SNMP_PollHost {
} inherits from Object;

// Some constants here for performance rather than looking them up...
const           sysUpTimeObjID = "1.3.6.1.2.1.1.3.0";
const           ifNumberObjID = "1.3.6.1.2.1.2.1.0";
const           ifTypeObjID = "1.3.6.1.2.1.2.2.1.3.";
const           ifOperStatusObjID = "1.3.6.1.2.1.2.2.1.8.";
const           ifInUcastPktsObjID = "1.3.6.1.2.1.2.2.1.11.";
const           ipAdEntAddrObjID = "1.3.6.1.2.1.4.20.1.1";
const           ipAdEntIfIndexObjID = "1.3.6.1.2.1.4.20.1.2";
const           ipAdEntNetMaskObjID = "1.3.6.1.2.1.4.20.1.3";
const           ipAdEntBcastAddrObjID = "1.3.6.1.2.1.4.20.1.4";
const           ipRouteIfIndexObjID = "1.3.6.1.2.1.4.21.1.2";
const           ipRouteNextHopObjID = "1.3.6.1.2.1.4.21.1.7";
const           ipNetToMediaNetAddressObjID = "1.3.6.1.2.1.4.22.1.3";
```

```
const           MAX_VARS_PER_REQUEST = 5;

SNMP_PollHost:create(oid master, oid hostInfo, string community,
        array(any) addresses)
{
        string          hostAddress;
        string          objID, ifNum;
        int             i, addr, end_block, count;
        oid             mibDirectory, sqeInterface;
        int             ok, rtt;

        set(oid)        obj_set;
        set(any)        result_set;
        array(any)      result;
        array(int)      addrTable;
        array(int)      addrIndex;
        int             addrCount;
        string          haltKey;
        int             haltKeyLength;

        obj_set = send "allInstances"("SNMP_QueryEngineInterface")
                        to RootObject;

        for sqeInterface in obj_set break;
        if (sqeInterface == nil) {
                write_cout("No SNMP Query Engine Interface on this host\n");
                exit;
        }
        obj_set = send "allInstances"("SNMP_MIB_Directory") to RootObject;

        for mibDirectory in obj_set break;
        if (mibDirectory == nil) {
                write_cout("No SNMP Query Engine Interface on this host\n");
                exit;
        }
        //
        // First, find interface that responds to SNMP
        //
        i = 0;
        addr = addresses[i];
        ok = 1;
        while (ok) {
                hostAddress = dottedAddress(addr);
                if (community == "PINGONLY") {
                        rtt = send "pingHost"(addr) to sqeInterface;
                        if (rtt != -1) {        // no SNMP response, but up
                                send "doneWithHost"(hostInfo, COMPLETED)
                                        to master;
                        } else {
                                send "doneWithHost"(hostInfo, NO_RESPONSE)
                                        to master;
                        }
                        exit;
                }
                result_set = send "getMIBvalue"(addr, community,
                        sysUpTimeObjID) to sqeInterface;

                if (result_set != nil) ok = 0;
                if (ok == 1) {  // no SNMP response...
                        rtt = send "pingHost"(addr) to sqeInterface;
                        if (rtt != -1) {        // no SNMP response, but up
                                send "doneWithHost"(hostInfo, NO_SNMP_RESPONSE)
                                        to master;
```

```
                        exit;
                }
                i += 1;
                addr = addresses[i];
                if (addr == nil) ok = 0;
        }
}
if (addr == nil) {      // complete failure!
        send "doneWithHost"(hostInfo, NO_RESPONSE) to master;
        exit;
}
hostAddress = dottedAddress(addr);
//
// Second, dump IP address table for ifIndex->IP address
// mapping for this host.
//
obj_set = emptySet;
obj_set += ipAdEntAddrObjID;
obj_set += ipAdEntIfIndexObjID;
ok = 1;
haltKey = ipAdEntAddrObjID + ".";
haltKeyLength = length(haltKey);
while (ok) {
        result_set = send "getNextMIBvalue"(addr, community,
                obj_set) to sqeInterface;
        if (result_set == nil) {
                rtt = send "pingHost"(addr) to sqeInterface;
                if (rtt != -1) {        // no SNMP response, but
                                        // up
                        send "doneWithHost"(hostInfo, NO_SNMP_RESPONSE)
                                to master;
                        exit;
                }
                send "doneWithHost"(hostInfo, NO_RESPONSE) to master;
                exit;
        }
        i = 0;
        obj_set = emptySet;
        ok = 0;         // only successfully query disproves
                        // this...
        for result in result_set {
                objID = result[0];
                obj_set += objID;
                if (i == 0) {   // ipAdEntAddr
                        objID = midstr(objID, 0, haltKeyLength);
                        if (objID == haltKey) { // still in table
                                ok = 1;
                                addrTable[addrCount] = result[2];
                        }
                } else if (i == 1) {    // ipAdEntIfIndex
                        if (ok == 1) {  // still in table
                                addrIndex[addrCount] = result[2];
                        }
                }
                i += 1;
        }
        if (ok == 1) addrCount += 1;
}
//
```

```
        // Third, dump interface table:  get interface status and
        // packet count.
        //
        count = 0;
        while (count < addrCount) {
                end_block = count + MAX_VARS_PER_REQUEST;
                if (end_block > addrCount) end_block = addrCount;
                obj_set = emptySet;
                // ask for interface type, status for each interface
                for (i = count; i < end_block; i += 1) {
                        ifNum = to_string(addrIndex[i]);
                        obj_set += ifOperStatusObjID + ifNum;
                        obj_set += ifInUcastPktsObjID + ifNum;
                }
                // we don't actually care what the results are
                // here...
                send "getMIBvalue"(addr, community,
                        obj_set) to sqeInterface from nil;
                count = end_block;      // next block of
                                        // MAX_VARS_PER_REQUEST
        }
        send "doneWithHost"(hostInfo, COMPLETED) to master;
}

SNMP_PollHost:delete()
{
}
```

nlp: A Network Printing Tool

Mark Fletcher – SAS Institute Inc.

ABSTRACT

The **nlp** system originated out of the need to make printers available to as many platforms as possible yet be as simple as possible. We studied several commercial printing systems and found them to be either too vendor specific or too complicated for our needs.

nlp is a TCP/IP client-server based model that utilizes a centralized database of all available printers. This database maps the name of a printer to the host that supplies the printing resource. Berkely's LPD protocol was chosen as our communication link between the client (the machine making the print request) and the server (where the job is queued and printed). LPD[1] was chosen because it was felt that it was a protocol used by more vendors than any other. Thus, any participating print server must run some form of **lpd**. **lpd**'s job is to receive a remote print request and hand jobs over to a local printing system (vendor supplied) for printing. **nlp** is therefore a transport agent between a client machine and another machine containing a printer resource acting much as **sendmail** does for Unix mail systems.

Introduction

SAS Institute Inc. is a company that utilizes a wide range of computer platforms including MVS, VM, VMS, Apollos, PCs, Macs, and various BSD and System V based Unix workstations. In the early years of SAS, printers were primarily found on mainframes and all printing was done in that environment. As other platforms matured and printers like the HP LaserJet began to infiltrate, there came the need to access virtually any printer from virtually any host. This paper describes the evolution of SAS printing and the development of the network printing facility called **nlp** which we designed and implemented to accomplish the goal of multi-platform environment printing.

In late February 1992, the Institute's software development base was moved from the Apollo workstation to the newly released HP 9000/700 series workstation. We felt that this was the perfect time to put in place a new printing system and try to leave behind the miles of "bailing wire" and "band-daids" of printer "interconnectivity" which had previously infested our platforms like some kind of virus.

At that time, printing between platforms involved **ftp**ing or **rsh**ing files to their destinations followed by **ftp**ing or **ftp**ing commands that were to perform the printing of those files. Many individuals developed their own printing scripts while others did it by hand each time. There was much duplication of effort. Many of the scripts quit working over time because of changes in network topologies while none of the scripts proved to be 100% reliable. Troubleshooting of printer related problems became incredibly difficult and completely within the realm of only a couple of people.

Presently, we have over 300 printers available to **nlp**. All these printers are accessible to MVS, VM, VMS, BSD Unix workstations (Suns, DECs, etc.) and our largest base of almost 800 HP 9000/720 workstations. Efforts are being made to port **nlp** to the PC and the Mac, but to date those platforms are not participating.

Most of our Unix based printers are attached to the HP workstations. This is a desirable configuration because we can utilize the built in configurable command line option capability and shell script nature of System V's printing system. We have built a very elaborate shell script structure around the printers attached to the HP's which allows us to do file type autodetection, fancy printing options, etc. Since the syntax of **nlp** is identical to the System V lp command, these options are available to **nlp** in exactly the same manner as they are with the local lp.

Life Before nlp

Like many companies, SAS Institute began in a mainframe environment. All software development, documentation, accounting, etc. was done on MVS. Naturally, any printed document was produced entirely in that environment. A very sophisticated printing system evolved on MVS; providing the capabilities for special kinds of forms, letter head, 2-up, 4-up, as well as a variety of fonts and page layouts.

Over the years, other computer systems began to join the SAS family: VM, VMS, Apollo, PC, Macintosh and a variety of Unix workstations. Each of these systems had their own printing facilities but

[1] The term LPD will be used throughout this document when referring to the Berkeley LPR/LPD protocol. **lpd** is used to mean the print server binary itself.

could only print to their own printers. There was, and still is, a strong need to get to MVS printers from any of these platforms. The need to get to any printer from any host has always been a legitimate concern of ours.

TCP/IP brought about connectivity between our major platforms. Users quickly realized that they could get files printed on another system by **ftp**ing them and then remotely logging in to that system to print the job. Soon, hundreds of user created tools sprang up to automate this process. Some became quite elaborate while most were "hard wired" to get a file from point A to point B. All this user creativity was a great service to the community but generated many problems:

1. There was much duplication of effort. A user would write a tool to get files printed on a specific printer then share that tool with other members of the department. Each user might modify the tool slightly to do something a little different or go to other printers. In a while, there might be ten or twelve similar versions of a tool spread across a hundred users.
2. Many tools were written that depended on the existence of particular host names or network topologies. The second a certain machine left the network or the topology changed, the tool broke. Users all over would be affected in some way.
3. Users often had no way to monitor or cancel their print jobs since tools sent the jobs potentially through two or three machines leaving them without a clue of where or how to look for them.
4. No one was responsible for overall connectivity. When someone's tool quit working, that person would probably call the Help Desk because they didn't know where they got the tool or the person who wrote the tool left for another job. The Help Desk would, of course, have no idea how to help the user and would end up calling around to see if anyone knew what to do. They usually didn't.

It was clear to everyone that a printing system that could span platform boundaries was needed.

Printing System Goals

With about ten years to realize what the printing problems were, it didn't take long to decide what goals were important in the design of a printing system. Our goals for a network printing system were:

1. Choose a protocol supported by the greatest number of platforms. We decided that this protocol should be Berkeley's LPD.[2] All the BSD machines supported this protocol as did the HP-UX machines. The VMS group had just purchased BSD software that supported it; as had the PC group. The MVS and VM groups located public domain versions of LPD and were beginning to test them on their systems.
2. We wanted to use a system's native print spooling software whenever possible. We were in the process of installing some 800 HP workstations at our site and planned to have a number of printers connected to these machines. These printers would primarily serve the users of the HP but we wanted anyone to be able to use them. We wanted to use the System V print system to drive these printers so we would not have to write our own. What we were really looking for was a transport agent that could adequately move print jobs between computer systems and interface with the printer host's native print system. It was also important to have the ability to query and cancel print jobs from any host.
3. It was important that print jobs took the shortest possible route to their destination. We wanted good performance and simplicity. The job should be spooled in as few places as possible. Under the old system, we had some print jobs that would get queued as many as three times.
4. To simplify administration, a data base should exist containing necessary information about any printer on the network. Important information would be the physical location of the printer, the type of printer, the options that printer would receive, the host that printer was attached to, and other details an administrator of that printer might care about. This data base should be centrally located so that it could be easily edited as printer changes occurred.
5. The user interface to the printer system should be uniform. That is, regardless of the platform, the command name, arguments and output would look the same.
6. A client host would need nothing more than a single binary (or set of binaries to perform the submit, status, and cancel functions). No local print queues, lookup tables, or printer data bases should exist on the local host.

The Design of nlp

The **nlp** system is designed around a TCP/IP client-server model. **nlp** is actually a transport agent between the user and a host's native printing system speaking LPD. It is responsible for getting a print job to the appropriate host which will in turn, hand the job over to its own local printing system. **nlp**

[2]The protocol is well defined in RFC-1179 and many documents have been published about it.

can connect to any host that runs some form of **lpd**. For any given platform, **lpd** performs three main tasks:

1. Listens for a TCP socket connection on port 515. Once the connection is established, **lpd** enters a server mode and waits for commands from its client.
2. Receives and responds to print requests from the client according to the LPD protocol defined in RFC-1179.
3. Translates LPD commands between **lpr** and the local printing system.

Currently, we have **lpds** running on MVS, VM, VMS, BSD, and HP-UX.

MVS Public domain program that interfaces with the MVS print system.

VM Public domain program that has been modified to submit print jobs to MVS for printing. Does not support query or cancel requests.

VMS Part of the Wollengong system. Interfaces to the native VMS printing system.

BSD The real thing.

HP-UX Part of the BSD compatibility software. The **lpd** binary is called **rlpdaemon** and interfaces to the System V print system.

The **nlp** user interface consists of three binaries:

nlp submits a job or jobs to a printer.

nlpstat displays the status of printer jobs and sites.

ncancel cancels jobs that have been queued.

The **nlp** binaries are command line compatible with the System V commands **lp**(1), **lpstat**(1), and **cancel**(1). **nlp**, **nlpstat**, and **ncancel** appear as **lp**, **lpstat**, and **cancel** to the user; and as **lpr**, **lpq**, and **lprm** to the connecting server.

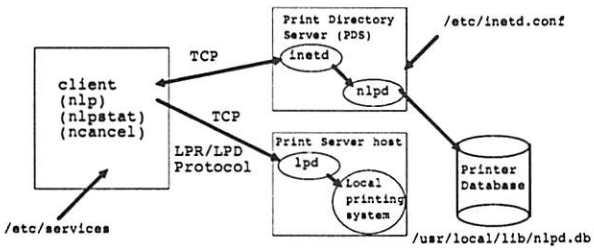


Figure 1: **nlp** I/O connections and files

When the user invokes **nlp**, **nlpstat**, or **ncancel**, a printer name must be supplied either in the environment variable LPDEST or with the command line option "-d". Before the client binary can connect to the correct print server host, it must consult a special machine called the *Print Directory Server*, PDS (see Figure 1). The PDS provides the client with (at the very least) the name of the host where the printer resides and the name given to that printer by the local printing system. For example:

```
someclient$ nlp -dhplj3si t.c
```

will first look up *hplj3si* on the PDS. Assume PDS returns *bert* as the printer host and *hplj* as the local printer name.

nlp then will create a TCP socket to port 515 on *bert* and send a print request to the printer named **hplj**. Upon successful response from *bert*, the contents of the file *t.c* is sent. *bert* then hands the job over to its local printing system which queues the job to *hplj*.

Two additional options have been added to **nlp** which allows the PDS to provide further information to the user.

nlp -l simply lists all the printer names contained in the PDS data base.

nlp -i *printername* Prints out all information contained in the PDS data base for the printer printername.

Typical information contained in the PDS data base for a printer might be:

- The host to send the job to. This is a host running **lpd** and a local printing system. The actual printer is usually attached directly to this machine.
- The brand name of the printer and the type of output it produces.
- The physical location of the printer (building, room number).
- A list of the options supported by the printer local print system.
- the languages the printer handles (e.g., text, Postscript, PCL).
- an example of sending **nlp** requests to this printer.
- any other documentation appropriate for this printer.

The PDS data base is kept in */usr/local/lib/nlp.db*. Each printer entry in the data base consists of a line of the form shown in Figure 2 followed by any number of "U" lines. "U" lines allow text in any format to be entered. **nlp -i** will simply print all "U" lines exactly as they appear in the data base.

An example */usr/local/lib/nlp.db* file containing a line printer and a laser printer is listed in Figure 3.

The **nlp** data base information commands would print information on these two printers. The output from these commands is shown in Figure 4.

```
P printername:hostname:local-printername:default-options
```

Figure 2: Printer entry

To prevent the **nlp** system from failing if the PDS is down, the data base is replicated across five machines. The machines are given aliases in **named** called *nlpserv[0-4]*. **nlp** will first try to contact *nlpserv0* for the data base, then *nlpserv1*, until **nlpserv4** fails in which case **nlp** exits with an error.

Because **nlp** appears as **lpr** to print server hosts, we started out with the **lpr** example listed in *Unix Network Programming*[3] and built the System V front end to it. The PDS gets invoked by **inetd** and requires a line in */etc/inetd.conf*.

[3]W. Richard Stevens (Prentice-Hall, Inc., 1990), pp. 543-554.

```
nlpserv0$ grep nlpd /etc/inetd.conf
nlpd stream tcp nowait root \
     /usr/local/etc/nlpd nlpd
```

Clients must contain an entry in the */etc/services* file supplying the port number to connect to on the PDS.

```
someclient$ grep nlpd /etc/services
nlpd 3003/tcp
```

The client will create a TCP socket to the port obtained in */etc/services* on *nlpserv0*. **inetd** will answer the request and invoke **nlpd** and attach its standard I/O to the newly created socket. **nlpd** will provide the data base information requested by the client. The information provided ranges from a host name, local printer name pair, to "U" entry

```
# Printer Directory Server data base
# Comrex 420 dot matrix line printer.
P c420:bert:lp:noheader
U Description: Comrex 420 dot matrix line printer.
U
U Location:    Room 123
U
U Host:        bert
U
U Options:
U              noheader       suppress burst page.
U              nlq            Near letter quality mode.
U
U Languages:   text only.
U
U Example:     nlp -dc420 -onlq memo
U                             Prints memo in near letter quality mode.
#
# HP Laserjet II laser printer
P hpljii:ernie:hpljii:
U Description: HP Laserjet II laser printer.
U
U Location:    Third floor lab
U
U Host:        ernie
U
U Options:
U              portrait       select portrait orientation.
U              landscape      select landscape orientation.
U              10       set pitch to 10 cpi.
U              12       set pitch to 12 cpi.
U
U Languages:   text and PCL.
U
U Example:     nlp -dhpljii -o'12 portrait' memo
U                            Prints memo at 12 characters per inch
U                            portrait.
```

Figure 3: Example Print Directory Server data base

information on one or all the printers in the data base.

The nlp Control File

For each file to be printed, RFC-1179 defines a *control* file in addition to the data file. See Figure 5.

Control Lines Used by nlp

nlp uses only a few of the control lines defined in the RFC. The "C" record is used (misused) to pass user defined options.

It is the responsibility of **lpd** to know to pick off the option string from the "C" record. The versions we have on the HP, VM, and MVS already do this so no modification was required.

Printers Attached to the HP System

We presently have about 30 printers attached to our HP 700 workstation platform. Most of these are HP Laserjet IIs and IIIs. There are a few dumb line printers and some special color printers and plotters. All are accessible via **nlp**.

```
someclient$ nlp -l
c420  hpljii
someclient$ nlp -i hpljii
Description: HP Laserjet II laser printer.
Location:    Third floor lab
Host:        ernie
Options:
           portrait     select portrait orientation.
           landscape    select landscape orientation.
           10           set pitch to 10 cpi.
           12           set pitch to 12 cpi.
Languages:   text and PCL.
Example:     nlp -dhpljii -o'12 portrait' memo
                  Prints memo at 12 characters per inch portrait.
```

Figure 4: **nlp** PDS information commands

```
Cclass\n        Sets the class name to be printed on the banner page.
Hhost\n         Specifies the host name where the print job originated.
Icount\n        Specifies the left margin indentation.
Jjobname\n      The name of the job.
Luser\n         The name of the submitting user.
Muser\n         Causes mail to be sent to user upon job completion
Nname\n         The name of the original file.
Pname\n         The UID of the submitting user.
Sdevice inode\n Records the device and inode of symbolic link.
Ttitle\n        Title.
Ufile\n         Unlink the file after printing.
Wwidth\n        Limit column to width when printing.
1file\n         Specifies the file for troff R font.
2file\n         Specifies the file for troff I font.
3file\n         Specifies the file for troff B font.
4file\n         Specifies the file for troff S font.
cfile\n         Plot CIF file.
dfile\n         Print DVI file.
ffile\n         Print as a plain ASCII file.
gfile\n         Print as data from Berkeley Unix plot library.
k               Reserved for use by Kerberized LPR clients.
lfile\n         Pass control characters unfiltered.
nfile\n         Print ditroff file.
```

Figure 5: RFC-1179 defined control lines

HP-UX is a System V based machine which uses the **lp** printing system. **lp** is much more flexible than Berkeley's **lpr** printing system because it supports generic options and uses a shell script (called an interface) to actually direct the printing. Generic options, entered via the "-o" command line option, allow highly customizable printer control. **rlpdaemon** gets these options from the "C" record in the control file.

To try to protect printers from improper file types and to try to make life easier for users, we developed a *language decoder* (similar to **file**(1)) which unless told to do otherwise, will interrogate a print job to determine if the file is an ASCII, Postscript, HP PCL, HP GL, etc. type file and print (or reject with a mail message) appropriately. For a Postscript printer, if a text file is sent to it, the *language decoder* will determine that the file is text and filter it through a text-to-Postscript translator and then queue the output to the printer. The text-to-Postscript program provides a number of control options and the user can specify any of these options via **nlp**'s "-o" command option.

The command shown in Figure 6 will send the job to the host corresponding to the printer name *hplj3si* which we assume to be an HP700. The **lp** interface will pass the document through the text-to-Postscript translator with the options "l" (landscape) and "s14" (14 point face). The resulting Postscript code is then queued to *hplj3si* for printing.

The control file **nlp** creates for this example is:

```
Hsomeclient
Pjoeuser
Cl s14
Ljoeuser
Jdocument.txt
fdfA038someclient
UdfA038someclient
Ndocument.txt
```

Using the interface script and the "-o" option capability of **lp**, we are able to create any kind of "look" to a printer that we want to. For this reason, we have tried to keep all Unix based printers on the HP.

Future Enhancements

There are a number of enhancements planned for **nlp** and even more requested by users. Here are some of them:

- Include more control lines defined by RFC-1179.
- Provide alternate print server sites. Print servers would be replicated so that if a server is down, the PDS would choose another. This works out well when printers are network attached instead of physically attached to particular machines. We are currently in the process of putting network cards in our Laserjet printers so this will likely be the next enhancement made to **nlp**.
- *nlp.allow* and *nlp.deny* files restricting user or client access to specific printers.
- Notification that a job has printed via an X windows based alarm.
- A means of issuing administrative commands for clearing, restarting, and rearranging the print queue of a local print system remotely through **nlp**.

Author Information

Mark Fletcher is a system administrator for the Unix support group at SAS Institute Inc. Reach him via U.S. Mail at SAS Institute Inc.; SAS Campus Drive; Cary NC 27513. Reach him electronically at mark@unx.sas.com.

```
someclient$ nlp -dhplj3si -o'l s14' document1.txt
```

Figure 6: Send job to hplj3si

H*host*\n	Specifies the host name where the print job originated.
P*name*\n	The UID of the submitting user.
C*options*\n	Provides a means of sending printer specific options.
L*user*\n	The name of the submitting user.
J*jobname*\n	The name of the job.
f*file*\n	Print as a plain ASCII file.
U*file*\n	Unlink the file after printing.
N*name*\n	The name of the original file.

Figure 7: Control lines used by **nlp**

NAME

nlp – network printing system.

SYNOPSIS

nlp [**–h**][**–l**][**–i** [**printer(s)**]] [**–d** | **–p** *printer*][*lp options*][*file...*]

nlp –h

Prints usage

nlp –l

Prints a list of available printers

nlp –i [*printer(s)*]

Prints information on all (or specified) printers

nlp [**–d** | **–p** *printer*] [*lp options*] [*filelist...*]

Submits file list (or standard input) to printer If *printer* is omitted, **LPDEST** environment variable is used

lp options:

–o*'options'*	Printer specific options
–s	Suppress "request id is ..." message
–t*title*	Title of print job to be placed on burst page

Other lp options:

–c, -m, -n *copies*, **-w**

These options are ignored and are for compatibility only

nlpstat [**–d** |**–p** *printer*][**–u**[*user(s)*]] [**–v**[*printer(s)*]] [**–hdrtacios**]

nlpstat –h

Prints usage

nlpstat –d

Name of default printer (contents of **LPDEST**)

nlpstat [**–d** |**–p** *printer(s)*] **–r**

Status of the remote system's print server (**lpd**)
NOTE: If no printer is specified, all printers are listed

nlpstat [**–d** | **–p***printer(s)*] **–t**

List of all jobs queued to printer(s) (default)

nlpstat [**–d** |**–p** *printer(s)*] **–a**

Long form of **–t**

nlpstat [**–d** |**–p** *printer(s)*] **–u**[*user(s)*]

All jobs owned by *user(s)*

nlpstat [**–d** |**–p** *printer(s)*] **–v**[*printer(s)*]

Prints name of system that *printer* is connected to

nlpstat [**–d** |**–p** *printer(s)*] **–i**[*printer(s)*]

Prints information about *printer(s)*

Included for compatibility with lp:

–o and **–s** — Same as **–t**
–c — Ignored by **nlp**

ncancel [**–a**] [**–h**] [**–d** |**–p** *printer*] [*job-id(s)*] [*user(s)*]

ncancel –h
Prints usage

ncancel [**–d** |**–p** *printer*] *job-number(s)*
Cancels specific jobs on *printer.*

ncancel –a [**–d** |**–p** *printer*]
Cancels all jobs owned by *user* on *printer.* If user is *root,* all jobs on *printer* are canceled. Must be *root* to cancel a job not owned by *user.*

ncancel [**–d** |**–p** *printer*] *user(s)*
Cancels all jobs owned by specified *user(s).* This command is used only by the *root* user.

NOTE: For compatibility with **cancel, ncancel** accepts the options **–e, –i,** and **–u** but ignores them

DESCRIPTION

The nlp system consists of three user comands:

nlp - submits jobs to a printer

nlpstat - displays the status of printer jobs and sites

ncancel - cancels jobs that have been queued

Note: The term **nlp** will be used in this document when referring to the **nlp system** and **nlp command** when referring to the **nlp** command itself.

nlp is by design, command line compatible with the System V commands **lp(1), lpstat(1),** and **cancel(1).**

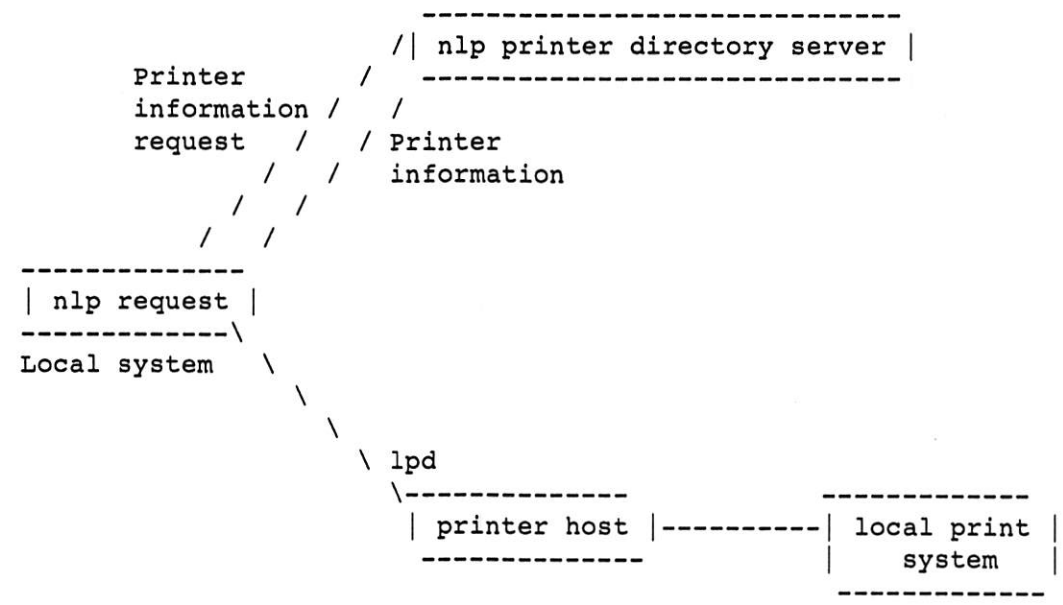
nlp is a transport system. It is responsible for getting a print job to the appropriate host which will in turn, hand the job over to its own local printing system. **nlp** is to printing what **sendmail(1M)** is to electronic mail. In order for a host to participate as an **nlp** print server, it must run the Berkeley **lpd** printing daemon. The underlying printing system is transparent to **nlp** since **nlp** communicates only with **lpd.** (See the diagram later in this document)

PRINTER DIRECTORY SERVER

nlp consults a **Printer Directory Server** to obtain information on all available printers. Typical information on a printer is:

* the brand name of the printer and the type of output it produces.
* the physical location of the printer (room number).
* the host to send the job to. This host runs an **lpd** daemon as well as a local printing system to control the printer. The printer is typically physically attached to this host and jobs to this printer are queued on this host.
* the options supported by the printer's local interface. This usually only applies to printers connected to System V machines (e.g. the HP9000). BSD based systems (e.g. Sun IV) have no capability to receive options.
* the languages the printer handles (e.g., text, Postscript, PCL).
* an example of sending **nlp** requests to this printer.
* other comments.

Printer data base information is obtained from the **Printer Directory Server** via the **–l** and **–i** options to **nlp.**



The nlp system

DEFAULT PRINTER

nlp does not maintain a system wide default printer. Instead, the user can establish a personal default printer by setting the **LPDEST** environment variable.

From the **C** shell:

% setenv LPDEST name-of-default-printer

From the **Korn** shell:

$ export LPDEST=name-of-default-printer

All subsequent **nlp** requests in which the **-d***printername* option is omitted will use the default printer destination.

The **nlpstat** command can display the current default printer using the **-d** option:

$ nlpstat -d
Default printer (LPDEST) = c2700j21

EXAMPLES

In the following examples, assume three types of printer:

a Postscript only printer called **cqms2h31**

an IBM mainframe printer called **c2700j21**

a Postscript/text printer called **chplje2**

Obtaining printer information on **cqms2h31, c2700j21,** and **chplje2:**

$ nlp -i cqms2h31

```
Description: QMS-PS 2000 high performance page printer.
Location:    H3109
Host:        sake.unx.sas.com
Options:     none
Languages:   Postscript
Example:     nlp -dcqms2h31 doc
Comments:
```

$ nlp -i c2700j21

```
Description: Xerox 2700 laser printer.
Location:    J298
Host:        sdcvm.vm.sas.com
Options:     As specified in "Data Center User's Guide".
Languages:   Text only.
Example:     nlp -dc2700j21 -o'UCS=let' doc
Comments:    This printer is accessed via sdcvm.
             No status or canceling features exist for this printer.
```

$ nlp -i chplje2

```
Description: Hewlett Packard LaserJet IIIsi.
Location:    E234
Host:        egon.unx.sas.com
Options:     Uses pstext options.  See man page on pstext.
Languages:   ASCII text and Postscript.
Example:     nlp -dchplje2 -o'-ld -s10' doc
Comments:    Automatically decides if file is text or Postscript by
             searching for "%!PS" at the start of the document.
             This is accomplished by the interface script.
```

```
# User sasxyz is using nlp on a system called jupiter
# The default printer is set to c2700j21
```

$ nlpstat -d

```
Default printer (LPDEST) = c2700j21
```

```
# Printing a text document on the default printer using the letter
# quality UCS option
```

$ nlp -o'UCS=let' document.txt

```
request id is 060jupiter (1 file)
```

```
# Using a text-to-Postscript translator called pstext to produce a
# "two up", 7 point, landscape Postscript version of a text file
# which is then input to nlp and sent to cqms2h31
```

$ pstext -s7 -ld test.c | nlp -dcqms2h31

request id is 061jupiter (1 file)

Using the Unix text formatting command pr(1) to pagenate a file
before printing it to the default printer

$ pr -f -l55 -hdocument.txt document.txt | nlp
request id is 062jupiter (1 file)

The chplje2 interface program automatically detects if a file is
Postscript or text. If the file is text, the text stream is passed
through pstext inside the interface before being sent to the printer.
The interface program accepts pstext options and passes them to
pstext when printing text files.

$ nlp -dchplje2 -o'-s7 -ld' test.c
request id is 063jupiter (1 file)

SUBMITTING JOBS

Print jobs can be either standard input (i.e., piped into **nlp** from other programs) or files. Any options available to a printer are listed in the printer data base.

Example of printing a file

$ nlp -dc2700j21 document.txt
request id is 064jupiter (1 file)

Example of printing standard output from another program

$ pr -f -w90 -l55 -htest.c test.c | nlp -dcqms2h31
request id is 065jupiter (1 file)

PRINTER STATUS

To display the status of the default printer:

$ nlpstat

Rank	Owner	Job	Files	Total Size
active	sasxyz	60	test.c	3432 bytes
1st	sasxyz	61	usage	1973 bytes
2nd	sasabc	62	passwd	34755 bytes
3rd	sasxyz	64	Makefile	681 bytes
4th	sasxyz	65	mport.src	1205 bytes

To display the status of cqms2h31:

$ nlpstat -cqms2h31

Rank	Owner	Job	Files	Total Size
active	sasxyz	81	ftp.1	72131 bytes

CANCELING JOBS

When a job is submitted, a job id is displayed to identify the job in the job queue:

$ nlp -dc2700j21 document.txt
request id is 066jupiter (1 file)

The numeric part of the *job id* displayed by **nlp** is used by **ncancel** to cancel a job. Both the data file and the corresponding control file are deleted on the printer host. If the job is successfully canceled, the data and control files are echoed. If the job does not exist, nothing is echoed:

Job **81jupiter** is deleted on host **sake :**

$ ncancel -dcqms2h31 81
sake: dfA081jupiter dequeued
sake: cfA081jupiter dequeued

Job **99** does not exist on **sake :**

$ ncancel -dcqms2h31 99

The user can cancel all jobs she/he owns using the **-a** option to **ncancel :**

$ ncancel -dchplje2 -a
egon: dfA036jupiter dequeued
egon: cfA036jupiter dequeued
egon: dfA039jupiter dequeued
egon: cfA039jupiter dequeued
egon: dfA040jupiter dequeued
egon: cfA040jupiter dequeued
egon: dfA041jupiter dequeued
egon: cfA041jupiter dequeued

CAVEATS

nlp is only as capable of managing printers as is the local printing system in which the printer is attached.

Examples:

BSD printer systems have a fixed option structure and are not configurable. Therefore, printer specific options are usually not available on these systems.

ncancel and **nlpstat** only work if the **lpd** server is capable of relaying these requests to the local printing system. The **lpd** running on **VM** and **MVS** do have any status or canceling support; thus **ncancel** and **nlpstat** will fail to any printer that is attached to these systems.

AUTHORS

The **nlp** system was written by Mark Fletcher and Dave Tilley.

SEE ALSO

lp(1), lpstat(1), cancel(1)

The USENIX Association

USENIX, the UNIX and Advanced Computing Systems professional and technical organization, is a not-for-profit membership association made up of systems researchers and developers, systems administrators, programmers, support staff, application developers and educators.

USENIX is dedicated to:

- fostering innovation and communicating research and technological developments,
- sharing ideas and experience, relevant to UNIX, UNIX-related and advanced computing systems, and
- providing a forum for the exercise of critical thought and airing of technical issues.

Founded in 1975, the Association sponsors two annual technical conferences and frequent symposia and workshops addressing special interest topics, such as C++, distributed systems, Mach, systems administration, and security. USENIX publishes proceedings of its meetings, a bi-monthly newsletter *;login:*, and a refereed technical quarterly, *Computing Systems*. The Association also actively participates in and reports on the activities of various ANSI, IEEE and ISO standards efforts.

There are four classes of membership in the Association: student, individual, institutional, and supporting, differentiated primarily by the fees paid and services provided. The supporting members of the USENIX Association are:

Digital Equipment Corporation
Frame Technology, Inc.
Matsushita Graphic Communication Systems, Inc.
mt Xinu
Open Software Foundation
Quality Micro Systems
Rational Corporation
Sun Microsystems, Inc.
Sybase, Inc.
UNIX System Laboratories, Inc.
UUNET Technologies, Inc.

For further information about membership or to order publications, contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710-2565
Telephone: 510/528-8649
Email: office@usenix.org
Fax: 510/548-5738

ISBN 1-880446-47-2